Journal of Computer Security 0 (0) 1
IOS Press

Mining Hierarchical Temporal Roles with Multiple Metrics [1]

Scott D.Stoller [a] ThangBui [a]
[a] *Department of Computer Science, Stony Brook University, U.S.A.*
E-mail: stoller@cs.stonybrook.edu, thang.bui@stonybrook.edu

Abstract. Temporal role-based access control (TRBAC) extends role-based access control to limit the times at which roles are enabled. This paper presents a new algorithm for mining high-quality TRBAC policies from timed ACLs (i.e., ACLs with time limits in the entries) and optionally user attribute information. Such algorithms have potential to significantly reduce the cost of migration from timed ACLs to TRBAC. The algorithm is parameterized by the policy quality metric. We consider multiple quality metrics, including number of roles, weighted structural complexity (a generalization of policy size), and (when user attribute information is available) interpretability, i.e., how well role membership can be characterized in terms of user attributes. Ours is the first TRBAC policy mining algorithm that produces hierarchical policies, and the first that optimizes weighted structural complexity or interpretability. In experiments with datasets based on real-world ACL policies, our algorithm is more effective than previous algorithms at optimizing policy quality.

Keywords: role mining, temporal 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). RBAC policy mining algorithms have potential to significantly reduce the cost of migration to RBAC, by partially automating the development of an RBAC policy from an access control list (ACL) policy and possibly other information, such as user attributes [4]. The most widely studied versions of the RBAC policy mining problem involve finding a minimum-size RBAC policy consistent with (i.e., equivalent to) given ACLs. When user attribute information is available, it is also important to maximize interpretability (or "meaning") of roles—in other words, to find roles whose membership can be characterized well in terms of user attributes. Interpretability is critical in practice. Researchers at HP Labs report "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" [2]. Algorithms for mining meaningful roles are described in, e.g., [10,16].

[1]This material is based on work supported in part by NSF under Grants CNS-1421893, CCF-1248184, and CCF-1414078, ONR under Grant N00014-15-1-2208, and AFOSR under Grant FA9550-14-1-0261. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

0926-227X/0-1900/$27.50 © 0 – IOS Press and the authors. All rights reserved

Temporal RBAC (TRBAC) extends RBAC to limit the times at which roles are enabled [1]. TRBAC supports an expressive notation, called *periodic expressions*, for expressing sets of time intervals during which a role is enabled. A role's permissions are available to members only while the role is enabled. This allows tighter enforcement of the principle of least privilege. Access control in many existing systems supports some form of groups or roles and some form of periodic temporal constraints. This includes LDAP-based directory servers, such as Oracle Unified Directory and Red Hat Directory Server, XACML-based Identity and Access Management (IAM) products, such as Axiomatics Policy Server, some other IAM products, such as NetIQ Access Manager, some cloud computing services, such as Joyent's Triton Compute Service, and many network routers and switches.

This paper presents an algorithm for mining hierarchical TRBAC policies. It is parameterized by a policy quality metric. We consider multiple policy quality metrics: number of roles, *weighted structural complexity* (WSC) [10], a generalization of syntactic policy size, *interpretability* (INT) [10,16], described briefly above, and a compound quality metric, denoted WSC-INT, that combines WSC and INT. Our algorithm does not require attribute data; attribute data, if available, is used only in the policy quality metric, if it considers interpretability. Our algorithm is the first TRBAC policy mining algorithm that produces hierarchical policies, and the first that optimizes WSC or interpretability.

Our algorithm is based on Xu and Stoller's elimination algorithm for RBAC mining [16] and some aspects of Mitra *et al.*'s pioneering generalized temporal role mining algorithm, which we call GTRM algorithm, for mining flat TRBAC policies (i.e., policies without role hierarchy) with minimal number of roles [7,8], which inspired our work. Our algorithm has four phases: (1) produce a set of candidate roles that contains initial roles (generated directly from the entitlements in the input) and roles created by intersecting initial roles, (2) merge candidate roles where possible, (3) organize the candidate roles into a role hierarchy, and (4) remove low-quality candidate roles (this is a greedy heuristic). The generated policy is not guaranteed to have optimal quality. Fundamentally, this is because the problem of finding an optimal policy is NP-complete (this follows from NP-completeness of the untimed version of the problem ([10]).

To evaluate the algorithm, we created datasets based on real-world ACL policies from HP, described in [2] and used in several evaluations of role mining algorithms, e.g., [10,16,8]. We could simply extend the ACLs with temporal information to create a temporal user-permission assignment (TUPA), and then mine a TRBAC policy from the TUPA and attribute data. However, it would be hard to evaluate the algorithm's effectiveness, because there is nothing with which to compare the quality of the mined policies. Therefore, we adopt a similar methodology as Mitra *et al.* [8]. For each ACL policy, we mine an RBAC policy from the ACLs and synthetic attribute data using Xu and Stoller's elimination algorithm [16], pseudorandomly extend the RBAC policy with temporal information numerous times to obtain TRBAC policies, expand the TRBAC policies into equivalent TUPAs, mine a TRBAC policy from each TUPA and the attribute data, and compare the average quality of the resulting TRBAC policies with the quality of the original TRBAC policy, with the goal that the former is at least as good as the latter.

We created two datasets, using different temporal information when extending RBAC policies to obtain TRBAC policies. For the first dataset, we use simple periodic expressions, each of which is a range of hours that implicitly repeats every day. We use the same time intervals as [8]. They are designed to cover various relationships between intervals, such as overlapping, consecutive, disjoint, and nested. For the second dataset, we use more complex periodic expressions based on a hospital staffing schedule. For both datasets, we use the same attribute data, namely, the high-fit synthetic attribute data for these ACL policies described in [16].

In experiments using number of roles as the policy quality metric, Mitra *et al.*'s GTRM algorithm, designed to minimize number of roles, produces 34% more roles than our algorithm, on average. In experiments using WSC-INT as the policy quality metric, our algorithm succeeds in finding the implicit structure in the TUPA, producing policies with comparable (for the first dataset) or moderately higher (for the second dataset) WSC and better interpretability, on average, compared with the original TRBAC policy.

Mitra *et al.* developed another temporal role mining algorithm, called the CO-TRAPMP-MVCL algorithm [9]. It minimizes a restricted variant of WSC based on the sizes of two components of the policy. In experiments using that variant as a policy quality metric, and using datasets created by Mitra *et al.*, our algorithm produces policies that are 41% smaller, on average, than the policies produced by the CO-TRAPMP-MVCL algorithm.

We explored the effect of different inheritance types on the quality of the mined policy and found that weakly restricted inheritance leads to policies with significantly better WSC and slightly better interpretability, on average. We experimentally evaluated the benefits of some design decisions and quantified the cost-quality trade-off provided by a parameter to our algorithm that limits the set of initial roles used in intersections in phase 1.

This paper is a revised and extended version of [12]. The main improvements are substitution of FastMiner for CompleteMiner when computing role intersections and an empirical justification for this, an improved metric for selecting a subset of initial roles for use in role intersections, more explanation and details of the algorithm, and more experiments, including an experimental comparison with Mitra *et al.*'s CO-TRAPMP-MVCL algorithm [9].

Section 2 provides background on TRBAC. Section 3 defines the policy mining problem. Section 4 presents our algorithm. Section 5 describes the datasets used in the experimental evaluation. Section 6 presents the results of the experimental evaluation. Section 7 discusses related work. Directions for future work include: mining TRBAC policies from operation logs, by extending work on mining RBAC policies from logs [11]; optimization of TRBAC policies, i.e., improving the quality of a TRBAC policy while minizing changes to it, by extending work on optimizing RBAC policies [14]; and mining temporal ABAC policies, by extending work on ABAC policy mining [17,6].

2. Background on TRBAC

An *RBAC policy* is a tuple $\langle User, Perm, Role, UA, PA, RH\rangle$, where $User$ is a set of users, $Perm$ is a set of permissions, $Role$ is a set of roles, $UA \subseteq User \times Role$ is the user-role assignment, $PA \subseteq Role \times Perm$ is the permission-role assignment, and $RH \subseteq Role \times Role$ is the role inheritance relation (also called the role hierarchy). 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'. A role r' is *junior to* role r if $r\,RH^+\,r'$, where RH^+ is the transitive closure of RH.

A *periodic expression* (PE) is a symbolic representation for an infinite set of time intervals. The formal definition of periodic expressions in [1,8] is standard and somewhat complicated; instead of repeating it, we give a brief intuitive version. A *calendar* is an infinite set of consecutive time intervals of the same duration; informally, it corresponds to a time unit, e.g., a day or an hour. A sequence of calendars $C_1, \ldots, C_n, C_d$ defines the sequence of time units used in a periodic expression, from larger to smaller. A periodic expression has the form $\sum_{k=1}^{n} O_k \cdot C_k \ \rhd\ d \cdot C_d$ where $O_1 = all$, O_k is a set of natural numbers or the special value all for $2 \leq k \leq n$, and d is a natural number. The first part of a PE (before

$\triangleright$) identifies the set of starting points of the intervals represented by the PE. The second part of the PE (after $\triangleright$) specifies the duration of each interval.

For example, consider the sequence of calendars Quadweeks, Weeks, Days, hours, where a Quadweek is four consecutive weeks—similar to a month, but with a uniform duration. The periodic expression [all · Quadweeks + {1,3} · Weeks + {1,2,3,4,5} · Days + {10} · Hours $\triangleright$ 8 · Hours] represents the set of time intervals starting at 9am (the time intervals in each calendar are indexed starting with 1, so for Hours, 1 denotes the hour starting at midnight, 2 denotes the hour starting at 1am, etc.) and ending at 5pm (since duration is 8 hours) of every weekday (assuming days of the week are indexed with 1=Monday) during the first and third weeks of every quadweek.

A *bounded periodic expression* (BPE) is a tuple $\langle [begin, end], pe \rangle$, where *begin* and *end* are date-times, and pe is a periodic expression. A BPE represents the set of time intervals represented by pe except limited to the interval $[begin, end]$.

A *BPE set* (BPES) is a set of BPEs. It represents the union of the sets of time intervals represented by its members

A *temporal RBAC (TRBAC) policy* is a tuple $\langle User, Perm, Role, UA, PA, RH, IT, REB \rangle$, where the first six components are the same as for an RBAC policy, IT is the inheritance type (described below), and REB is the role enabling base (REB), which specifies when roles are enabled [1]. Bertino *et al.* allow the REB to specify various conditions and events that enabled or disable a role. Like Mitra *et al.* [8,9], we are interested only in temporal conditions and therefore consider a limited form of REB, which we call a *role-time assignment*. Specifically, a role-time assignment TA maps each role to a BPES. A role r is enabled during the set of time intervals represented by $TA(r)$. A REB can easily be constructed from a role-time assignment, so an RBAC policy with temporal conditions represented by a role-time assignment instead of a REB can also be considered a TRBAC policy.

We consider two types of inheritance [5]. In both cases, a senior role r inherits permissions from each of its junior roles r'. With *weakly restricted inheritance*, denoted by $IT = \mathrm{WR}$, a permission inherited from r' is available to members of r during the time intervals specified by $TA(r)$. With *strongly restricted inheritance*, denoted by $IT = \mathrm{SR}$, a permission inherited from r' is available to members of r during the time intervals specified by $TA(r')$.

A *temporal user-permission assignment (TUPA)* is a set of triples of the form $\langle u, p, bpes \rangle$, where u is a user, p is a permission, and $bpes$ is a BPES. We refer to such a triple as an *entitlement triple*. Such a triple means that u has permission p during the set of time intervals represented by $bpes$. A TUPA should contain at most one entitlement triple for each user-permission pair. A TUPA can therefore be regarded as a dictionary that maps user-permission pairs to BPESs.

The meaning of a role r in a TRBAC policy π, denoted $[\![r]\!]_\pi$, is a TUPA that expresses the entitlements granted by r, taking inheritance into account. The meaning $[\![\pi]\!]$ of a TRBAC policy π is a TUPA that expresses the entitlements granted by π.

3. The Relaxed TRBAC Policy Mining Problem

A *policy quality metric* is a function from TRBAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this might seem counter-intuitive at first glance, but it is natural for metrics such as policy size. We define three basic policy quality metrics and then consider combinations of them.

Number of roles is a simplistic but traditional policy quality metric.

Weighted Structural Complexity (WSC) is a generalization of policy size [10]. We adapt WSC to TRBAC. For a TRBAC policy π of the above form, the WSC of π is defined by $\mathrm{WSC}(\pi) = w_1|Role| + w_2|UA| + w_3|PA| + w_4|RH| + w_5\mathrm{WSC}(TA)$, where the w_i are user-specified weights, $|s|$ is the size (cardinality) of set s, and $\mathrm{WSC}(TA)$ is the sum of the sizes of the BPESs in TA. The size of a BPES is the sum of the sizes of the BPEs in it. The size of a BPE is the size of the PE in it (the beginning and ending date-times have fixed size, so we ignore them). The size of a PE is the sum of the sizes of the sets in it plus 1 for the duration, with the special value all counted as a set of size 1.

Interpretability is a policy quality metric that measures how well role membership can be characterized in terms of user attributes. *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. 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 is defined by $\mathrm{mismatch}(e, U) = |\,[\![e]\!] \ominus U|$, 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 $\min_{e\in E} \mathrm{mismatch}(e, \mathrm{asgndU}(r))$, where E is the set of all attribute expressions, and $\mathrm{asgndU}(r) = \{u \mid \langle u, r\rangle \in UA\}$. An attribute expression e that minimizes the attribute mismatch of role r is called a *best-fit attribute expression for* r. Intuitively, it is the most accurate possible "explanation" (characterization) of r's membership using the given attribute data; it can be shown to users to help them understand the role. We define policy interpretability INT as the sum over roles of attribute mismatch, i.e., $\mathrm{INT}(\pi) = \sum_{r\in Role} \mathrm{AM}(r)$.

Compound policy quality metrics take multiple aspects of policy quality into account. We combine metrics by Cartesian product, with lexicographic order on the tuples. Lexicographic order means $\langle x_1, y_1\rangle < \langle x_1, y\rangle$ iff $x_1 < x_2$ or $x_1 = x_2 \wedge y_1 < y_2$. Weighted sums of policy quality metrics could also be used. Let WSC-INT$(\pi) = \langle \mathrm{WSC}(\pi), \mathrm{INT}(\pi)\rangle$.

A TRBAC policy π is *consistent* with a TUPA T if they grant the same permissions to the same users for the same sets of time intervals. When the given TUPA contains noise, it is desirable to weaken this requirement. A TRBAC policy π is ϵ-*consistent* with a TUPA T, where ϵ is a natural number, if they grant the same permissions to the same users for the same sets of time intervals, except that, for at most ϵ entitlement triples $\langle u, p, bpes\rangle$ in T, the policy π either does not grant p to u or grants p to u at fewer times than $bpes$ [8]. Note that consistency is a special case of ϵ-consistency, corresponding to $\epsilon = 0$.

The *relaxed TRBAC policy mining problem* is: given a TUPA T, policy quality metric Q_{pol}, and consistency threshold ϵ, find a TRBAC policy π that is ϵ-consistent with T and has the best quality, according to Q_{pol}, among policies ϵ-consistent with T. Auxiliary information used by the policy quality metric, e.g., user-attribute data, is implicitly considered to be part of Q_{pol} in this definition. Note that the temporal part of T strongly influences π, even using WSC with $w_5 = 0$, because it determines how entitlements can be grouped in roles.

We refer to this as the *relaxed* TRBAC policy mining problem, because of the relaxed consistency requirement; Mitra *et al.* refer to it as the *generalized* TRBAC policy mining problem.

Suggested role assignments for new users. If attribute data is available, the system can compute and store a best-fit attribute expression e_r for each role r. When a new user u is added, the system can suggest that u be made a member of the roles for which u satisfies the best-fit attribute expression, and it presents these suggested roles in ascending order of attribute mismatch. This reduces the administrative effort involved in assigning roles to new users.

```
R_init = new Set()
asgndP_0 = new Dictionary()
asgndU_0 = new Dictionary()
TA = new Dictionary()
for u in U
  for ⟨P, bpes⟩ in permBPES(u, T)
    addRole(R_init, {u}, P, bpes)
    for bpe in bpes
      addRole(R_init, {u}, P, {bpe})

permBPES(u, T) =
{⟨P, bpes⟩ | (∃p.⟨u, p, bpes⟩ ∈ T)
                ∧ P = {p | ⟨u, p, bpes'⟩ ∈ T
                              ∧bpes ⊑ bpes'}}
```

```
function addRole(R, U, P, bpes)
// if there is an existing role with permissions P and
// BPES bpes, add users in U to it, else create new
// role with users U, permissions P, and BPES bpes.
if U, P, or bpes is empty
  return
if ∃ r in R s.t. asgndP_0(r) = P ∧ TA(r) = bpes
  asgndU_0(r).addAll(U)
else
  r = new Role()
  asgndP_0(r) = P
  asgndU_0(r) = U
  TA(r) = bpes
  R.add(r)
```

Fig. 1. Phase 1.1: Generate initial roles. "s.t." abbreviates "such that".

4. TRBAC Policy Mining Algorithm

Inputs to the algorithm are the TUPA T, the type of inheritance IT to use in the generated policy, the consistency threshold ϵ, and the policy quality metric Q_{pol}. While reading the TUPA, our algorithm attempts to simplify the BPES in each triple by merging BPEs in it that represent sets of overlapping or consecutive time intervals; this is done in the same way as in case (2b) of Phase 2, described below.

In traditional RBAC and TRBAC notation, roles are identifiers (not objects), and separate relations such as UA (not object attributes) provide information about them. Similarly, in our pseudocode, roles have no attributes; instead, dictionaries map roles to relevant information.

Our pseudocode uses the following notation for sets and dictionaries. "new Set()" and "new Dictionary()" create an empty set and empty dictionary, respectively. The methods of a set s include $s.\mathrm{add}(x)$ to add an element x, $s.\mathrm{remove}(x)$ to remove an element x, $s.\mathrm{addAll}(x)$ to add all elements of set s_2, and $s.\mathrm{copy}(x)$ to create a copy of x. The statement $d(k) = v$ updates dictionary d to map key k to value v. The expression $d(k)$ returns the value that dictionary d associates with key k; it is used only in contexts where d contains an entry for k.

Phase 1: Generate roles. Phase 1 generates initial roles and then creates additional candidate roles by intersecting sets of initial roles.

Phase 1.1: Generate initial roles. Pseudocode for generating initial roles appears in Figure 1. It uses a semantic containment relation $\sqsubseteq$ on PEs, BPEs, and BPESs: $x_1 \sqsubseteq x_2$ iff the set of time instants represented by x_1 is a subset of the set of time instants represented by x_2. Note that, for BPESs $bpes_1$ and $bpes_2$, $bpes_1 \sqsubseteq bpes_2$ may hold even if $bpes_1 \subseteq bpes_2$ does not hold. The function permBPES groups together the set of permissions P that a user u has for exactly the same BPES $bpes$ or a BPES $bpes'$ that semantically contains $bpes$. An initial role is created with user u, the resulting set of permissions P, and time assignment $bpes$. In addition, for each BPE bpe in $bpes$, we create an initial role with user u, permissions P, and time assignment $\{bpe\}$.

Phase 1.2: Intersect roles. Phase 1.2 starts to construct a set R_{cand} of candidate roles, by adding to R_{cand} all of the initial roles in R_{init} and all non-empty intersections of all pairs of initial roles. In other

```
for r in R_init
  R_init.remove(r)
  R_cand.add(r)
  for r' in R_init
    P = asgndP_0(r) ∩ asgndP_0(r')
    bpes = TA(r) ⊓ TA(r')
    if P and bpes are non-empty
      addRole(R_cand, asgndU_0(r) ∪ asgndU_0(r'), P, bpes)
```

Fig. 2. Phase 1.2: Intersect roles

words, for each pair of initial roles, if the intersection of their permission sets is a non-empty set P, and the intersection of their BPESs is a non-empty BPES $bpes$, then create a candidate role with permissions P, BPES $bpes$, and the union of their user sets. BPESs are intersected semantically, not syntactically; for example, if $bpes_1$ represents 9am-5pm on Mondays and Wednesdays, and $bpes_2$ represents 1pm-2pm on Mondays and Fridays, then their intersection is a BPES that represents 1pm-2pm on Mondays. This phase is similar to role intersection in FastMiner [15]. Pseudocode appears in Figure 2. The function $\sqcap$ denotes semantic intersection of BPESs; in other words, $bpes_1 \sqcap bpes_2$ is a BPES that represents the set of time instants represented by $bpes_1$ and $bpes_2$.

This phase is expensive for large datasets. To reduce the cost, we allow role intersections to be limited to a subset of the initial roles containing the roles mostly likely to produce useful intersections. To support a flexible trade-off between cost (running time) and policy quality, we introduce a parameter that controls the size of the subset.

The subset is characterized using a new role quality metric, called the *usefulness-for-intersection metric* (UI metric). It is a weighted sum of four quantities relevant to the usefulness of a role r in intersections: role size (sum of number of users, number of permissions, and the WSC of the BPES), $\mathrm{covEntit}(r)$ (defined below), permission popularity (sum over the permissions p of r of the fraction of initial roles having permission p), and PE popularity (sum over the PEs pe in r's BPES of the fraction of initial roles having pe in its BPES). For example, consider the set of roles $\{r_1, r_2, r_3\}$, where r_1 has permissions $\{p_1, p_2\}$ and enabled time $\{pe_1\}$, r_2 has permissions $\{p_1\}$ and enabled time $\{pe_1\}$, and r_3 has permissions $\{p_4\}$ and enabled time $\{pe_2, pe_3\}$ (user assignments are irrelevant hence omitted). The permission popularity of r_1 is $\frac{2}{3} + \frac{1}{3} = 1$, of r_2 is $\frac{2}{3}$, and of r_3 is $\frac{1}{3}$. The PE popularity of r_1 is $\frac{2}{3}$, of r_2 is $\frac{2}{3}$, and of r_3 is $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$.

We used a Support Vector Machine (SVM) to find the weights that maximize the UI metric's effectiveness as a classifier for whether an initial role is "useful for intersections", i.e., is used in an intersection that contributes to the final policy, either directly or via merges. We extended our system to keep track of which initial roles are useful for intersections, ran the extended system on one small policy (domino), and trained the SVM on the resulting data. The resulting weights are -2.7357, -1.6484, 2.3417, and -0.6017, respectively. The signs of the parameters show that, for example, roles with smaller size and more popular permissions are more useful in intersections.

To control the cost-quality trade-off, we introduce a parameter RIC (mnemonic for "role intersection cutoff") that ranges between 0 and 1, sort the roles by the usefulness-for-intersection metric, and use only roles in the top RIC in intersections. For example, $\mathrm{RIC} = 0.3$ means that only roles whose values of the UI metric are in the top (*i.e.*, largest) 30% are used in intersections.

Phase 2: Merge roles. Phase 2 merges candidate roles to produce a revised set of candidate roles. It uses the following types of merges. (1) If candidate roles r and r' have the same set of users U and the same BPES $bpes$, then they are replaced with a new role with users U, permissions $\mathrm{asgndP}_0(r) \cup \mathrm{asgndP}_0(r')$, and BPES $bpes$, unless a role with those permissions and that BPES already exists, in which case the users U are added to it. (2) If candidate roles r and r' have the same users U and same permissions P, then they are replaced with a new role with users U, permissions P, and BPES $bpes(r) \sqcup bpes(r')$, unless a role with those permissions and that BPES already exists, in which case the users U are added to it. Pseudocode appears in Figure 3. The function $\sqcup$ denotes semantic union of BPESs; in other words, $bpes_1 \sqcup bpes_2$ is a BPES that represents the set of time instants represented by $bpes_1$ or $bpes_2$. We distinguish two sub-cases. (2a) If $bpes_1$ and $bpes_2$ represent disjoint sets of time intervals, then $bpes_1 \sqcup bpes_2$ is simply $bpes_1 \cup bpes_2$. (2b) If $bpes_1$ and $bpes_2$ represent sets of overlapping or consecutive time intervals, then BPEs in them are merged, if possible, to simplify the result. For example, if $bpes_1$ represents 9am-noon on weekdays, and $bpes_2$ denotes noon-5pm on weekdays, then $bpes_1 \sqcup bpes_2$ contains a single BPE denoting 9am-5pm on weekdays.

Phase 3: Construct role hierarchy. Phase 3 organizes the candidate roles into a role hierarchy with full inheritance. A TRBAC policy has *full inheritance* if every two roles that can be related by the inheritance relation are related by it, i.e., $\forall r, r' \in R.\ [\![r]\!]_\pi \supseteq [\![r']\!]_\pi \Rightarrow \langle r, r' \rangle \in RH^*$. Guo *et al.* call this property *completeness* in the context of RBAC [3]. We always generate policies with full inheritance, even though relaxing this requirement would allow our algorithms to achieve better policy quality in some cases, because in the absence of other information, all of these possible inheritance relationships are equally plausible, and removing any of them risks removing some that are semantically meaningful and desirable.

Phase 3.1: Compute inheritance. Phase 3.1 determines inheritance relationships between candidate roles, based on the requirement of full inheritance. Function $\mathrm{isAncestorFullInher}(r', r)$ tests whether r' is an ancestor of r with full inheritance; if $IT = \mathrm{WR}$, the function avoids inheritance relationships that would lead to cycles in the role hierarchy.

$$\begin{aligned}&\mathrm{isAncestorFullInher}(r', r) = \\ &\ \mathrm{asgndP}_0(r') \subseteq \mathrm{asgndP}_0(r) \wedge \mathrm{asgndU}_0(r) \subseteq \mathrm{asgndU}_0(r') \\ &\ \wedge (IT = \mathrm{SR} \Rightarrow TA(r') \sqsubseteq TA(r)) \\ &\ \wedge (IT = \mathrm{WR} \Rightarrow \neg(\mathrm{asgndP}_0(r) \subset \mathrm{asgndP}_0(r') \wedge \mathrm{asgndU}_0(r') \subset \mathrm{asgndU}_0(r)))\end{aligned}$$

This function is called for every pair of candidate roles. If $\mathrm{isAncestorFullInher}(r', r)$ is true, and there is no role between r' and r in the role hierarchy (i.e., no role r'' such that $\mathrm{isAncestorFullInher}(r', r'')$ and $\mathrm{isAncestorFullInher}(r'', r)$), then r' is a parent of r. This phase produces dictionaries $parents$ and $children$, such that $parents(r)$ and $children(r)$ are the sets of parents and children of r, respectively. Pseudocode appears in Figure 4.

Phase 3.2: Compute assigned users and permissions. Phase 3.2 computes the directly assigned users $\mathrm{asgndU}(r)$ and directly assigned permissions $\mathrm{asgndP}(r)$ of each role r, by removing inherited users and permissions from the role's originally assigned users $\mathrm{asgndU}_0(r)$ and originally assigned permissions $\mathrm{asgndP}_0(r)$. Pseudocode appears in Figure 5.

Phase 4: Remove roles. Phase 4 removes roles from the candidate role hierarchy if the removal preserves ϵ-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 sets of directly assigned users and permissions of other roles are expanded to contain users and permissions that they previously inherited from r.

```
R_vis = new Set()
for r in R_cand
  R_vis.add(r)
  for r' in R_cand \ R_vis
    mergeIfSameMemberBPES(R_cand, r, r')
    mergeIfSameMemberPerm(R_cand, r, r')

function mergeIfSameMemberBPES(R_cand, r, r')
if asgndU_0(r) = asgndU_0(r')
    ∧ TA(r) = TA(r')
  if asgndP_0(r) ⊆ asgndP_0(r')
    // merging r and r' yields r', so just remove r
    R_cand.remove(r)
  else if asgndP_0(r') ⊆ asgndP_0(r)
    // merging r and r' yields r, so just remove r'
    R_cand.remove(r')
  else
    P = asgndP_0(r) ∪ asgndP_0(r')
    if ∃ r'' in R_cand s.t. asgndP_0(r'') = P
                  ∧ TA(r'') = TA(r)
      asgndU_0(r'').addAll(asgndU_0(r))
    else
      r'' = new Role()
      asgndU_0(r'') = asgndU_0(r)
      asgndP_0(r'') = P
      TA(r'') = TA(r)
      R_cand.add(r)
    R_cand.remove(r)
    R_cand.remove(r')
```

```
function mergeIfSameMemberPerm(R_cand, r, r')
if asgndU_0(r) = asgndU_0(r')
    ∧ asgndP_0(r) = asgndP_0(r')
  if TA(r) ⊑ TA(r')
    R_cand.remove(r)
  else if TA(r') ⊑ TA(r)
    R_cand.remove(r')
  else
    bpes = TA(r) ⊔ TA(r')
    if ∃ r'' in R s.t. asgndP_0(r'') = asgndP_0(r)
                  ∧ TA(r'') = bpes
      asgndU_0(r'').addAll(asgndU_0(r))
    else
      r'' = new Role()
      asgndU_0(r'') = asgndU_0(r)
      asgndP_0(r'') = asgndP_0(r)
      TA(r'') = bpes
      R_cand.add(r)
    R_cand.remove(r)
    R_cand.remove(r')
```

Fig. 3. Phase 2: Merge roles.

The order in which roles are considered for removal affects the final result. 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. We use a role quality metric that is a temporal variant of the role quality metric in [16] that gave the best results in their experiments. We define some auxiliary functions then role quality.

The *redundancy* of a role r measures how many other roles also cover the entitlement triples covered by r. We say that a role r *covers* an entitlement triple t if $t \in [\![r]\!]_\pi$. Removing a role with higher redundancy is less likely to prevent subsequent removal of other roles, so we eliminate roles with higher redundancy first. The redundancy of role r, denoted $\mathrm{redun}(r)$, is the negative of the minimum, over entitlement triples $\langle u, p, bpes\rangle$ covered by r, of the number of removable roles that cover $\langle u, p, bpes\rangle$ (we take the negative so that roles with more redundancy have lower quality). A role is *removable* in policy π, denoted $\mathrm{removable}(r)$ (the policy is an implicit argument), if the policy obtained by removing r is ϵ-consistent

```
parents =new Dictionary()
children =new Dictionary()
for r in R_cand
  parents(r) = new Set()
  children(r) = new Set()
for r in R_cand
 for r' in R_cand \ {r}
  if isAncestorFullInher(r', r)
   // check whether r' is a parent or a more distant ancestor of r
   if ¬∃ r'' in parents(r) s.t. isAncestorFullInher(r', r'')
    // r' is a parent of r, based on roles considered so far.
    // a subsequent role could be placed between them.
    parents(r).add(r')
    // remove parents of r that are also parents of r'.
    for r'' in parents(r) \ {r'}
     if isAncestorFullInher(r'', r')
      parents(r).remove(r'')
  if isAncestorFullInher(r, r')
   // check whether r' is a child or more distant descendant of r
   if ¬∃ r'' in children(r) s.t. isAncestorFullInher(r'', r')
    // r' is a child of r, based on roles considered so far.
    // a subsequent role could be placed between them.
    children(r).add(r')
    // remove children of r that are also children of r'.
    for r'' in children(r) \ {r'}
     if isAncestorFullInher(r', r'')
      children(r).remove(r'')
```

Fig. 4. Phase 3.1: Determine inheritance relationships.

with T.

$$\mathrm{redun}(\langle u, p, bpes\rangle) = |\{r \in R_{\mathrm{cand}} \mid \langle u, p, bpes'\rangle \in [\![r]\!]_\pi \ \wedge bpes \sqsubseteq bpes' \wedge \mathrm{removable}(r)\}|$$

$$\mathrm{redun}(r) = -\min_{t \in [\![r]\!]_\pi} (redun(t))$$

The *clustered size* of a role r measures how many entitlements are covered by r and how well they are clustered. A first attempt at formulating this metric (ignoring clustering) might be as the fraction of entitlement triples in T that are covered by r. As discussed in [16], it is better for the covered entitlement triples to be "clustered" on (i.e., associated with) fewer users rather than being spread across many users. The clustered size of r is defined to equal the fraction of the entitlements of r's members that are covered by r. In the temporal case, each entitlement triple $\langle u, p, bpes\rangle$ is weighted by the fraction of the time represented $bpes$ that is covered by $TA(r)$.

$$\mathrm{covEntit}(r) = \sum_{\substack{u \in \mathrm{asgndU}(r) \\ p \in \mathrm{asgndP}(r)}} \frac{\mathrm{dur}(TA(r))}{\mathrm{dur}(T(u,p))} \qquad \mathrm{clsSz}(r) = \frac{\mathrm{covEntit}(r)}{|\mathrm{entitlements}(\mathrm{asgndU}(r), T)|}$$

```
for r in R_cand
  inheritedU = ⋃_{r' in children(r)} asgndU_0(r')
  asgndU(r) = asgndU_0(r).copy().removeAll(inheritedU)
  if IT=WR
    inheritedP = ⋃_{r' in parents(r)} asgndP_0(r')
    asgndP(r) = asgndP_0(r).copy().removeAll(inheritedP)
  if IT=SR
    asgndP(r) = asgndP_0(r).copy()
    for p in asgndP_0(r)
      // inherBPES is the BPES with which p is inherited by r
      inherBPES = ⊔_{r' ∈ parents(r)} TA(r')
      // if inherBPES equals TA(r), then p does not need to be directly assigned, i.e., p is inherited.
      if TA(r) = inherBPES
        asgndP.remove(p)
```

Fig. 5. Phase 3.2: Compute directly assigned users and directly assigned permissions.

where $T(u,p)$ is the BPES $bpes$ such that $\langle u, p, bpes\rangle \in T$, $\mathrm{dur}(bpes)$ is the fraction of one time unit in calendar C_1 that is covered by $bpes$, and $\mathrm{entitlements}(U,T)$ is the set of entitlement triples in T for a user in U. For example, if the sequence of calendars is $C_1 = \mathrm{Year}, \ldots, C_n = \mathrm{Hour}, C_d = \mathrm{Hour}$, and $bpes$ is 9am-5pm every day, then $\mathrm{dur}(bpes) = 1/3$, since $bpes$ covers 1/3 of the time in a year.

Our role quality metric is $Q_{role}(r) = \langle \mathrm{redun}(r), \mathrm{clsSz}(r)\rangle$, with lexicographic order on the tuples.

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 TRBAC policy resulting from the removal is related to the quality Q of the current TRBAC 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.

Pseudocode for removing roles appears in Figure 6. It repeatedly tries to remove all removable roles, until none of the attempted removals succeeds in improving the policy quality. The policy π is an implicit argument to auxiliary functions such as $\mathrm{removeRole}$ and $\mathrm{addRole}$. Function $\mathrm{addRole}(r)$ adds role r to the candidate role hierarchy: inheritance relations involving r are added, and the assigned users and assigned permissions of r's newly acquired ancestors and descendants are adjusted by removing inherited users and permissions, in a similar way as in the construction of the role hierarchy in Phase 3. Removing a role r and then restoring r using $\mathrm{addRole}$ leaves the policy unchanged.

When testing whether ϵ-consistency is violated, it is sufficient to check the size of $T \setminus [\![\pi]\!]$. It is unnecessary to consider $[\![\pi]\!] \setminus T$, because it is always empty; to see this, note that $[\![\pi]\!]$ equals T at the beginning of Phase 4, and Phase 4 only removes roles, which can only decrease $[\![\pi]\!]$.

The following auxiliary functions are used in $\mathrm{removeRole}$. isDescendant(r,r') holds if r is a descendant of r', as determined by following the parent-child relations in the $children$ dictionary. The set of authorized users of r, denoted $\mathrm{authU}(r)$, is the set of users in $\mathrm{asgndU}(r)$ or $\mathrm{asgndU}(r')$ for some r' senior to r; this is the same as in RBAC. The notion of authorized permissions must be defined differently in TRBAC than RBAC, because, with strongly-restricted inheritance, the inherited permissions of a role r may be associated with BPESs different than $TA(r)$. With strongly-restricted inheritance, the set of authorized permissions of r, denoted $\mathrm{authP}(r)$, is the set of permission-BPES pairs $\langle p, bpes\rangle$ such that (1) each directly assigned permission of r is paired with $TA(r)$ and (2) each permission p inherited

```
π = policy from Phase 3
q = Q_pol(π)
workL = list of removable roles in π
changed = true
while ¬empty(workL) ∧ changed
 sort workL in ascending order by Q_role
 changed = false
 for r in workL
  removeRole(r)
  // if ϵ-consistency is violated,
  // restore r.
  if |T \ [[π]] | > ϵ
   addRole(r)
   workL.remove(r)
  else
   // if policy quality improved,
   // keep the change.
   if Q_pol(π) < δq
    changed = true
    q = Q_pol(π)
    workL.remove(r)
   else
    // undo the change, i.e., restore r
    addRole(r)
```

```
function removeRole(r)
for parent in parents(r)
 // remove r from its parents
 children(parent).remove(r)
 for child in children(r)
  // if child is not a descendant of parent
  // after removing r, add an inheritance
  // edge between child and parent.
  if ¬ isDescendant(child,parent)
   children(parent).add(child)
   parents(child).add(parent)
 for u in asgndU(r)
  // if u is not authorized to parent after
  // removing r, add u to assigned users
  // of parent.
if u ∉ authU(parent)
   asgndU(parent).add(u)
for child in children(r)
 parents(child).remove(r)
 for p in asgndP(r)
  // if p is not fully authorized to child
  // after removing r, add p to assigned
  // permissions of child.
  if ⟨r, TA(child)⟩¬ ∈ authP(child)
   asgndP(child).add(p)
R_cand.remove(r)
```

Fig. 6. Phase 4: Remove roles.

by r is paired with the semantic union of the BPESs of the junior roles from which it is inherited. With weakly-restricted inheritance, $\mathrm{authP}(r)$ is the set of permission-BPES pairs $\langle p, TA(r)\rangle$ such that p is in $\mathrm{asgndP}(r)$ or $\mathrm{asgndP}(r')$ for some r' junior to r; we use a set of pairs for uniformity with the case of strongly-restricted inheritance.

5. Datasets

We generated two datasets based on real-world ACL policies from HP, described in [2], and the high-fit synthetic attribute data for these ACL policies described in [16]; see those references for more information about the ACL policies and attribute data. Briefly, the ACL policies are named americas_small, apj, domino, emea, firewall1, firewall2, and healthcare. The synthetic attribute data is generated pseudo-randomly, using statistical distributions based on statistical summaries of some real-world attribute data, to make the synthetic data more realistic. The number of attributes ranges from 20 to 50, depending on the policy size. The type of attribute values is unimportant (the only operation used by our algorithm on attribute values is equality), so we simply use natural numbers for the values of all attributes.

As outlined in Section 1, for each ACL policy, we mine an RBAC policy from the ACLs and the attribute data using Xu and Stoller's elimination algorithm [16], and pseudorandomly extend the RBAC

policy with temporal information several times to obtain TRBAC policies. For each ACL policy except americas_small, we create 30 TRBAC policies. For americas_small, which is larger, we create only 10 TRBAC policies, to reduce the running time of the experiments. We extend the RBAC policies in two ways, using different temporal information.

Dataset with simple PEs. A *simple PE* is a range of hours (e.g., 9am-5pm) that implicitly repeats every day. We define the WSC of a simple PE to be 1. This dataset uses the same simple PEs as in [8], namely, $[6, 11]$, $[7, 10]$, $[8, 9]$, $[8, 11]$, $[9, 11]$, $[10, 11]$, $[10, 12]$, $[11, 13]$, $[14, 15]$, $[16, 17]$. These PEs are designed to cover various relationships between intervals, such as overlapping, consecutive, disjoint, and nested. We choose the number of PEs in each BPES pseudorandomly using a similar probability distribution as in [8], namely, $pr(1) = 0.78, pr(2) = 0.2, pr(3) = 0.02$. We choose the specific PEs in each BPES pseudorandomly using a uniform distribution.

Dataset with complex PEs. For this dataset, we use periodic expressions based on a hospital staffing schedule, based on discussions with the Director of Timekeeping at Stony Brook University Hospital. The periodic expressions are not taken directly from the hospital's staffing schedule, but they reflect its general nature. The schedule does not repeat every week, but rather every few weeks, because weekend duty rotates. Clinicians may work 3 days/week for 12 hours/day starting at 7am or 7pm, or 5 days/week for 8.5 hours/day starting at 7am, 3pm, or 11pm. The probabilities of these work schedules are 0.144, 0.094, 0.284, 0.284, and 0.194, respectively. We create two instances of each of these five types of work schedules, by pseudorandomly choosing the appropriate number of days of the week in each of the four weeks of a Quadweek, using a uniform distribution. Each BPES is based on exactly one of the resulting 10 work schedules. Multiple PEs are needed to represent work schedules that wrap around calendar units; for example, a 7pm-7am shift is represented using two PEs, with time intervals 7pm-midnight and midnight-7am. The PEs are based on the following sequence of calendars: C_1=Quadweeks, C_2=Days, C_3=Hours, C_d=Hours. The days in a Quadweek are numbered 1..28. Including Week in the sequence of calendars is not helpful, because most workers' schedules do not repeat on a weekly basis. For example, consider a clinician who works 3 days/week for 12 hours/day starting at 7am, working Mon,Wed,Fri during the first and second weeks of a quadweek, and Tue,Thu,Sat during the third and fourth weeks. Assuming weeks start on Monday, this schedule is represented by the PE [$all \cdot$ Quadweeks + $\{1, 3, 5, 8, 10, 12, 16, 18, 20, 23, 25, 27\} \cdot$ Days + $\{8\} \cdot$ Hours $\triangleright$ 12 $\cdot$ Hours].

6. Evaluation

The experimental methodology is outlined in Section 1. All experiments use quality change tolerance $\delta = 1.001$ (this value gave the best results for the experiments in [16]), $\epsilon = 0$, and $w_i = 1$ for all weights in WSC. The policy quality metric is WSC-INT, and the inheritance type is weakly restricted, except where specified otherwise.

Our Java code and datasets are available at `http://www.cs.stonybrook.edu/~stoller/software/`. Periodic expressions are an abstract data type with two implementations: (1) simple PEs, as defined in Section 5, and implemented as pairs of integers, and (2) (general) PEs, as defined in Section 2, and implemented as arrays of arrays of integers. These implementations are used in the experiments in Sections 6.1 and 6.2, respectively. Running times include the cost of an end-to-end correctness check that checks equivalence of the input TUPA and the meaning of the mined TRBAC policy; the average cost is about 7% of the running time. The experiments were run on a Lenovo IdeaCentre K430 with a 3.4 GHz Intel Core i7-3770 CPU.

Dataset	Original Policy			Mined Policy						Time	Avg $\|R\|$	
	WSC		INT	WSC			INT				Our	
	μ	σ		μ	σ	CI	μ	σ	CI	μ	Alg	GTRM
americas_small	6975	7.5	189	7098	71	27	138	6	2.2	48:42	296	
apj	4879	10.0	385	4813	16	5.9	384	3.4	1.3	0:15	470	527
domino	449	2.5	23	450	9	3	18	1.9	0.70	0:01	29	40
emea	3929	4.4	32	4065	80	30	32	0	-	0:41	99	115
firewall1	1533	4.1	48	1603	80	30	37	3.4	1.3	1:07	93	130
firewall2	960	1.4	7	963	7.2	2.7	4	1.2	0.44	0:02	12	17
healthcare	168	1.4	14	165	1.6	0.6	12	1.2	0.44	0:01	16	25

Fig. 7. Results of experiments with simple PEs.

6.1. Experiments using dataset with simple PEs

All experiments on this simple PEs dataset use role intersection cutoff $\mathrm{RIC} = 1$.

Comparison of original and mined policies. Figure 7 shows detailed results from experiments on this dataset. In the column headings, μ is mean, σ is standard deviation, CI is half-width of 95% confidence interval using Student's t-distribution, and time is the average running time in minutes:seconds. There is no standard deviation column for INT, because interpretability is unaffected by the role-time assignment and hence is the same for all TRBAC policies generated by extending the same RBAC policy. Ignore the last 2 columns for now. The averages and standard deviations are computed over the TRBAC policies created by extending each RBAC policy. The WSC of the mined TRBAC policy ranges from about 2% lower (for healthcare) to about 5% higher (for firewall1) than the WSC of the original TRBAC policy. The interpretability of the mined policy ranges from about 40% lower (for firewall-2) to about 1% lower (for apj) than the interpretability of the original TRBAC policy. On average over the seven policies, the WSC is about 0.5% higher, and the interpretability is about 19% lower. Thus, our algorithm succeeds in finding the implicit structure in the TUPA and producing a policy with comparable WSC and better interpretability, on average, than the original TRBAC policy.

Comparison of FastMiner and CompleteMiner. In Phase 1.2 (Intersect roles), instead of the FastMiner approach of computing intersections only for pairs of initial roles, we could instead adopt the CompleteMiner approach of computing intersections for all subsets of initial roles [15]. We ran our algorithm, modified to use CompleterMiner, on our simple PE dataset, omitting emea and americas_small because of their longer running times. Figure 8 shows the results using FastMiner and CompleteMiner. Surprisingly, CompleteMiner did not improve policy quality: it increased the average WSC by 4% on average, ranging from 0.2% (for firewall2) to 11% (for domino), and it increased (worsened) the average INT by 10% on average, ranging from 1% (for apj) to 19% (for firewall1). Although one might expect that generating additional candidate roles would only improve the quality of the final policy, the role selection phase uses imperfect heuristics, so additional candidate roles sometimes lead to decreases in policy quality. Not surprisingly, CompleteMiner is slower: it increased the average running time by 160% on average, ranging from 15% for firewall2 to 201% for apj.

Comparison of inheritance types. We ran our algorithm again on the same dataset with all policies except americas_small, specifying strongly restricted inheritance for the mined policies. This caused a significant increase in the WSC of the mined policies. The percentage increase averages 51% and

Dataset	WSC				INT				Time	
	CM		FM		CM		FM		CM	FM
	μ	σ	μ	σ	μ	σ	μ	σ	μ	μ
healthcare	168	4	165	1.6	14	0.4	12	1.2	0:01	0:01
firewall2	966	9	963	7.2	5	1.0	4	1.2	0:02	0:02
firewall1	1661	64	1603	80	44	3.7	37	3.4	1:13	1:07
domino	500	71	450	9	21	1.5	18	1.9	0:01	0:01
apj	4828	21	4813	16	388	3.7	384	3.4	0:46	0:15

Fig. 8. Results of experiments with Complete Miner (CM) and Fast Miner (FM).

ranges from 6% (for apj) to 105% (for firewall1 and healthcare). Intuitively, the reason for the increase is that, with strongly restricted inheritance, the temporal information associated with directly assigned and inherited permissions may be different, and this may prevent removing inherited permissions from a role's directly assigned permissions. Inheritance type has less effect on the average INT, increasing (worsening) it by about 3% on average.

Evaluation of choice of initial roles. Recall from Section 4 that the definition of permBPES in Figure 1 uses the condition $bpes \sqsubseteq bpes'$ in order to include in each initial role the permissions that the user has for a BPES $bpes'$ that semantically contains $bpes$. A more obvious alternative is to require $bpes = bpes'$ and thereby include only the permissions that the user has for exactly the same BPES $bpes$. Let permBPES$^-$ denote that variant of permBPES. We evaluated the benefit of using permBPES by running our algorithm, modified to use permBPES$^-$ instead of permBPES, for all policies in the simple PE dataset except the largest one, americas_small, due to its longer running time. This change increased the average WSC by 37% on average, ranging from 13% (for apj) to 85% (for healthcare). It increased (worsened) the average INT by 50% on average, ranging from 9% (for apj) to 100% (for emea). The average running time decreased by 61% on average, ranging from 31% slower (for firewall2) (the only policy for which the modified algorithm was slower) to 94% faster (for emea).

The policy quality benefit of permBPES over permBPES$^-$ can also be demonstrated with a simple example. Consider the input TUPA $T = \{\langle u_1, p_1,\ 10am{-}5pm\rangle,\ \langle u_1,\ p_2, 10am{-}noon\rangle,\ \langle u_1, p_3, noon{-}5pm\rangle\}$. Our algorithm generates a policy with 2 roles and WSC 8; one role has permissions $\{p_1, p_2\}$ during 10am-noon, and the other role has permissions $\{p_1, p_3\}$ during noon-5pm. The variant of our algorithm that uses permREB$^-$ instead of permBPES generates a policy with 3 roles, each corresponding to one element of the TUPA, and with WSC 9. Mitra *et al.*'s GTRM algorithm [8] also produces that policy, as expected, since its construction of initial roles is more similar to permBPES$^-$ than permBPES. Mitra *et al.*'s CO-TRAPMP-MVCL algorithm [9] may produce either of these policies, depending on the value of a parameter, namely, the threshold θ for degree of overlap.

We also evaluated the effect of using both permBPES and permBPES$^-$, *i.e.*, of replacing the call $\mathrm{permBPES}(u, T)$ with $\mathrm{permBPES}(u, T) \cup \mathrm{permBPES}^-(u, T)$. This change increased the average WSC by 0.1% and the average INT by 0.2%. It also increased the average running time by 22% on average, ranging from 7% faster (for firewall1) to 60% slower (for domino).

We considered reducing the cost of Phase 1.1 by removing the first call to addRole. Note that Mitra *et al.*'s algorithm does not include an analogue of this call. This change increased the average WSC by 9% on average over the policies used in this experiment (all except americas_small), ranging from 7% (for emea and firewall2) to 10% (for domino). It increased (worsened) the average INT by 8% on average over those policies, ranging from 2% (for firewall2) to 12% (for firewall1).

Dataset	Original Policy			Mined Policy						RIC	Time
	WSC		INT	WSC			INT				
	μ	σ		μ	σ	CI	μ	σ	CI		μ
apj	16836	159	385	16879	165	205	383	3.1	3.8	1	72:42
domino	1156	49	23	1256	64	24	16	2.0	0.7	1	0:34
emea	5975	99	32	7309	354	440	32	0	0	0.4	41:24
firewall1	3712	97	48	6534	509	190	46.8	3.9	4.9	0.4	324:48
firewall2	1269	37	7	1316	56	21	3.4	1.3	0.5	1	1:00
healthcare	560	35	14	592	38	48	8.8	1.2	1.5	1	11:00

Fig. 9. Results of experiments with complex PEs.

Comparison with Mitra et al.'s GTRM algorithm. We ran Mitra *et al.*'s GTRM algorithm [8], and our algorithm with number of roles as policy quality metric (because GTRM algorithm optimizes this metric), on our dataset with simple PEs. Their code supports only simple PEs, so we used only the simple PE dataset in the comparison. Their code, implemented in C, gave an error ("malloc: ...: pointer being freed was not allocated") on some TRBAC policies generated for emea and firewall1; we ignored those results. Their code did not run correctly on americas_small, so we omitted it from this comparison.

The last two columns of Figure 7 show the numbers of roles generated by the two algorithms. Standard deviations are omitted to save space but are small: on average, 3% of the mean, for both algorithms. The GTRM algorithm produces 34% more roles than ours, on average. Our algorithm produces hierarchical policies, and their algorithm produces flat policies, but this does not affect the number of roles. There are many other differences between the algorithms, discussed in Section 7, which contribute to the difference in results. The above paragraph on evaluation of choice of initial roles describes two experiments that explore differences between our algorithm and the GTRM algorithm and quantify the benefit of those differences. The effects of some other differences between the two algorithms, such as the use of elimination *vs.* selection in Phase 4, were investigated in the untimed case in [16] and likely have a similar impact here.

6.2. Experiments using dataset with complex PEs

Comparison of original and mined policies. Figure 9 shows detailed results from experiments on this dataset. The original TRBAC policies here have higher WSC than the ones in Section 6.1, because complex PEs have higher WSC than simple PEs. We averaged over 30 TRBAC policies each for domino and firewall2, and (to reduce the running time of the experiments) 5 TRBAC policies each for the others. For emea and firewall1, we use $\mathrm{RIC} = 0.4$ instead of $\mathrm{RIC} = 1$ to reduce the running time. The average WSC of the mined TRBAC policies ranges from 0.3% higher (for apj) to 76% higher (for firewall1) than the WSC of the original TRBAC policy. The average interpretability of the mined TRBAC policies ranges from 52% lower (for firewall2) to 0.5% lower (for apj) than the interpretability of the original TRBAC policy. On average over the four policies for which we use $\mathrm{RIC} = 1$, the WSC is 5% higher, and the interpretability is 30% lower. On average over the two policies for which we use $\mathrm{RIC} = 0.4$, the WSC is 49% higher, and the interpretability is 1% lower. On average over all six policies, the WSC is 19% higher, and the interpretability is 20% lower. Thus, our algorithm finds most of the implicit structure in the TUPA and produces a policy with moderately higher WSC and better interpretability, on average, than the original TRBAC policy. The results can be improved by using larger RIC, at the expense of higher running time.

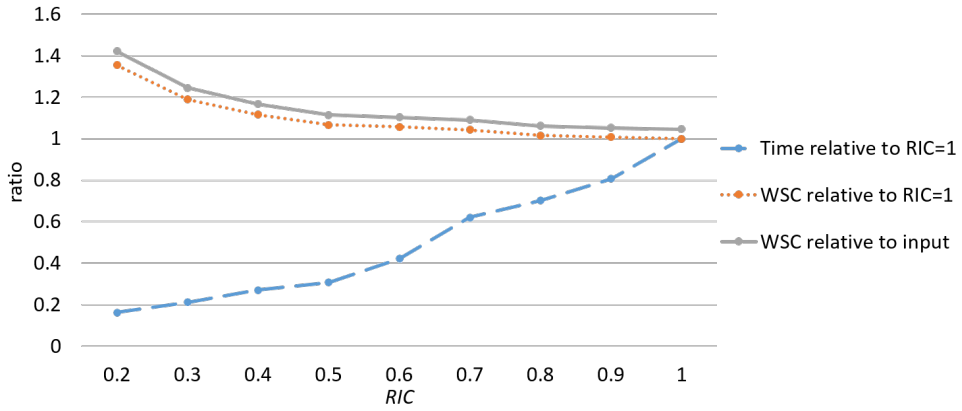

Fig. 10. Relative running time and relative WSC as functions of RIC.

The higher running times, compared to the dataset with simple PEs, are due primarily to the larger number of candidate roles created by role intersection (there are more overlaps between BPESs in this dataset), and secondarily to the larger overhead of manipulating more complex PEs.

Benefit of general PEs. PEs can be translated into sets of simple PEs. For example, the set of PEs {[all · Weeks + {1,2,7} · Days + {1} · Hours ▷ 8 · Hours]} can be translated to the set of simple PEs {[1,9], [25,33], [145,153]}. However, PEs are generally more compact and efficient than simple PEs. For example, in experiments with the healthcare, domino, and firewall2 policies, which have the smallest WSCs among our example policies, using this translation and simple PEs was about 5x, 12x, and 14x slower, respectively, than using general PEs.

Cost-benefit trade-off from role intersection cutoff. We investigated the cost-benefit trade-off when varying the role intersection cutoff RIC. Figure 10 shows running time and WSC as functions of RIC, averaged over apj, domino, firewall2, healthcare, which are four of the smaller policies. The trade-off is favorable: as RIC decreases, running time decreases much more rapidly than WSC increases. For example, at RIC $= 0.5$, running time is 70% lower than with RIC $= 1$, and WSC is only 11% higher.

Benefit of new RIC *metric.* We evaluated the advantage of the userfulness-for-intersection (UI) metric in Section 4 over covEntit, which is the UI metric in our DBSec 2016 paper [12]. In experiments with apj, domino, emea, firewall2, and healthcare, for RIC $= 0.4$, mining with covEntit as the UI metric takes 2.5 times longer and produces policies with 17% higher WSC than mining with the new UI metric, on average over those policies.

6.3. Comparison with Mitra et al.'s CO-TRAPMP-MVCL algorithm

Mitra *et al.*'s CO-TRAPMP-MVCL algorithm, called the CTR algorithm for brevity, minimizes a variant of WSC, called cumulative overhead of temporal roles and permissions (CO-TRAP), defined by $w_{TA}.|TA| + w_{PA}.|PA|$, where w_{TA} and w_{PA} are user-specified weights [9]. Mitra *et al.* use w_{PA} = w_{TA} = 1 for their experiments, and we use the same values. In these experiments, we run our algorithm with the following weights for WSC: $w_1 = 0$, $w_2 = 0$, $w_3 = 1$, $w_4 = 0$, $w_5 = 1$. This means the WSC equals $|PA| + |TA|$, the same as CO-TRAP. CO-TRAP is designed for non-hierarchical policies, so we flatten the hierarchical policies produced by our algorithm and then compute CO-TRAP for the flattened policies. Flattening transforms a hierarchical TRBAC policy into an equivalent non-hierarchical policy,

Dataset	c25_o75			c50_o50			c75_o25			c100			o100		
	Our Algorithm		CTR	Our Algorithm		CTR	Our Algorithm		CTR	Our Algorithm		CTR	Our Algorithm		CTR
	μ	σ	μ	μ	σ	μ	μ	σ	μ	μ	σ	μ	μ	σ	μ
healthcare	92	17	279	124	33	287	142	39	281	83	13	265	191	40	283
domino	420	43	627	391	25	631	402	37	632	405	35	625	405	38	634
apj	1813	11	2375	1817	9	2524	1832	9	2605	1799	8	2303	1849	5	2640
firewall1	1109	120	2819	1142	88	3370	1202	110	3432	1076	89	2704	1276	94	3353
firewall2	602	1	941	612	48	941	603	2	941	602	2	947	604	5	944
emea	5542	192	7245	5634	201	7245	5751	193	7245	5385	176	7245	5856	159	7245
americas_large	67288	964	94515	69077	795	96020	68108	878	96797	60734	1029	91971	62393	1012	97110
americas_small	3616	247	9563	3834	213	10052	4358	264	10446	3321	180	8567	4296	228	10618

Fig. 11. Comparison of our algorithm and the CO-TRAPMP-MVCL (a.k.a. CTR) algorithm using the CO-TRAP metric.

by adding direct user-role assignments for all role memberships that are inherited in the hierarchical policy, and then removing the role hierarchy. Coincidentally, flattening leaves TA and PA unchanged, so we get the same result regardless of whether we compute CO-TRAP for the hierarchical policy or the flattened policy.

Dataset. Our experimental comparison with the CTR algorithm uses the datasets generated by Mitra *et al.* for their experiments with CTR algorithm described in [9]. It is based on the same real-world ACL policies from HP as our datasets described in Section 5. It contains TRBAC policies generated by mining non-temporal RBAC policies using Ene *et al.*'s algorithm [2], and then extending them with synthetic temporal information containing simple PEs. First, they create 10 sets of contained time intervals (the intervals in each set are totally ordered by the subset relation) and 10 sets of overlapping time intervals (every pair of intervals in each set has a non-empty intersection). They create a role-time assignment by pseudorandomly associating some of these time intervals with each role, selecting from the sets of contained time intervals and overlapping time intervals with probability d and $1 - d$, respectively, where d is a parameter of the generation process. They generate five datasets, each for a different value of d: 1, 0.75, 0.50, 0.25, and 0. These datasets are denoted c100, c75o25, c50o50, c25o75, and o100, respectively. Each dataset contains 30 TRBAC policies with different pseudorandom role-time assignments.

Results. Figure 11 shows the average μ and standard deviation σ of CO-TRAP for policies generated by our algorithm, and average CO-TRAP for policies generated by CTR algorithm as reported in [8, Table 6]. The average CO-TRAP for policies generated by our algorithm ranges from 68% lower (for healthcare c100) to 19% lower (for emea o100) than the corresponding results for the CTR algorithm. On average over all five datasets for all eight ACL policies, results for policies generated by our algorithm are 41% lower than results for policies generated by the CTR algorithm. Thus, our algorithm is significantly more effective than the CTR algorithm at minimizing CO-TRAP.

It took less than 2 minutes to run our algorithm for all 30 TRBAC policies generated from each of the ACL policies healthcare, domino, firewall2, and emea. It took less than 2 minutes to run our algorithm for each TRBAC policy generated from apj, firewall1, and americas_large (an ACL policy from HP not used in the datasets described in Section 5). It took approximately 24 minutes to run experiments for each TRBAC policy generated from americas_small. Mitra *et al.* report that "each individual run took no more than 24 minutes" [9]. Although these measurements are from experiments on different hardware and software platforms (our algorithm is implemented in Java, and CTR algorithm is implemented in C), they suggest that running times of our algorithm and CTR algorithm are comparable.

7. Related Work

We discuss related work on TRBAC policy mining and then related work on RBAC mining. Role mining (for RBAC or TRBAC) is also reminiscent of some other data mining problems, but algorithms for those other problems are not well suited to role mining. For example, association rule mining algorithms are designed to find rules that are probabilistic in nature and are supported by statistically strong evidence. They are not designed to produce a set of rules strictly consistent with the input that completely covers the input and is minimum-sized among such sets of rules.

7.1. Related Work on TRBAC Policy Mining

Mitra *et al.* define a version of the TRBAC policy mining problem, called the generalized temporal role mining (GTRM) problem, based on minimizing the number of roles. They present an algorithm, which we call the GTRM algorithm, for approximately solving this problem [8]. It is an improved version of their earlier work [7].

Mitra *et al.* also define another version of the TRBAC policy mining problem, called cumulative overhead of temporal roles and permissions minimization problem (CO-TRAPMP), based on minimizing the CO-TRAP metric described in Section 6.1. They present another algorithm, called CO-TRAPMP-MVCL, for heuristically solving this problem [9].

Our algorithm is more flexible than the GTRM and CO-TRAPMP-MVCL algorithms, because our algorithm can optimize a variety of metrics, including WSC and interpretability. The importance of interpretability is discussed in Section 1. WSC is a more general measure of policy size than number of roles or CO-TRAP and can more accurately reflect expected administrative cost. For example, the average number of role assignments per user is a measure of expected administrative effort for adding a new user [13], and this can be reflected in WSC by giving appropriate weight to the size of the user-role assignment. Neither number of roles nor CO-TRAP take the size of the user-role assignment into account.

Our algorithm produces hierarchical TRBAC policies. The GTRM and CO-TRAPMP-MVCL algorithms produce flat TRBAC policies. Role hierarchy is a well-known feature of RBAC that can significantly reduce policy size and administrative effort by avoiding redundancy in the policy.

Our algorithm and the GTRM algorithm have a similar high-level structure: they both (1) create a large set of candidate roles based on the input TUPA, (2) merge some candidate roles, and then (3) select a subset of the candidate roles to include in the final policy. The algorithms also have many differences. Some differences are related to policy quality metric and role hierarchy, as discussed above. Some other differences are: (1) Our algorithm determines which candidate roles to include in the final policy by elimination of low-quality roles, instead of selection of high-quality roles. We showed that elimination gives better results in the untimed case [16]. (2) Our algorithm creates more initial roles than the GTRM algorithm. The benefit of creating these additional initial roles is shown in Section 6.1. The GTRM algorithm creates *unit roles*, which are similar to our initial roles but have only one permission. In particular, an initial role created by the second call to addRole in our algorithm is a unit role only when P is a singleton set and $\mathrm{permBPES}(u, T) = \mathrm{permBPES}^{-}(u, T)$; we not expect this to be a common case, since most temporal roles have multiple permissions. (3) Our algorithm performs fewer types of intersections than the GTRM algorithm. The GTRM algorithm performs five types of intersections, corresponding to r_a, r_b, r_c, r_d, r_e in [8, Algorithm 1]. Our algorithm performs only intersections corresponding to r_a. We omit r_b and r_c because they may create PEs with time intervals that do not appear in the input TUPA and

are not intuitive to security administrators. We omit r_d and r_e because Phase 3 would merge those roles back into the roles from which they were created. (4) Our algorithm performs more merges; specifically, the GTRM algorithm does not include case (2a) of the merge in Phase 2 of our algorithm.

The CO-TRAPMP-MVCL algorithm has a different high-level structure than our algorithm: roughly speaking, it (1) repeatedly generates a small set of candidate roles based on the current set of uncovered triples and adds the best one among them to the policy, and then (2) merges some roles. In the experiments in Section 6.3, our algorithm produces higher-quality policies than CO-TRAPMP-MVCL algorithm, as measured using the CO-TRAP metric which the CO-TRAPMP-MVCL algorithm is designed to optimize.

Our implementation supports periodic expressions for specifying temporal information, while Mitra *et al.*'s implementations of the GTRM and CO-TRAPMP-MVCL algorithms support only ranges of hours that implicitly repeat every day. Design and implementation of operations on sets of PEs is non-trivial. This includes operations such as testing whether one set of PEs covers all of the time instants covered by another set of PEs, and handling numerous corner cases, such as time intervals that wrap around calendar units (e.g., a 7pm-7am work shift).

7.2. Related Work on RBAC Mining

A survey of work on RBAC mining appears in [4]. The most closely related work is Xu and Stoller's elimination algorithm [16]. We chose it as the starting point for design of our algorithm, because in the experiments in [16], it optimizes WSC more effectively than Hierarchical Miner [10] and the Graph Optimisation role mining algorithm [18], while simultaneously achieving good interpretability, and it optimizes WSCA, an interpretability metric defined in [10], more effectively than Attribute Miner [10].

Our algorithm retains the overall structure of the elimination algorithm but differs in several ways, due to the complexities created by considering time. Our algorithm introduces more kinds of candidate roles than the elimination algorithm, because it needs to consider grouping permissions that are enabled for the same time or a subset of the time of other permissions. Our algorithm attempts to merge candidate roles; the elimination algorithm does not. Construction of the role hierarchy is significantly more complicated than in the elimination algorithm; for example, with strongly restricted inheritance, a permission p can be inherited by a role r from multiple junior roles with different BPESs, which may together cover all or only part of the time that p is available in r. This also complicates adjustment of the role hierarchy when removing candidate roles. The role quality metric used to select roles for removal is more complicated, to give preference to roles that cover permissions for more times.

We thank the authors of [8,9]—Barsha Mitra, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya—for sharing their code and datasets with us and helping us understand their work.

References

[1] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[2] 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)*, pages 1–10. ACM, 2008.

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

[4] S. Hachana, N. Cuppens-Boulahia, and F. Cuppens. Role mining to assist authorization governance: How far have we gone? *International Journal of Secure Software Engineering*, 3(4):45–64, October-December 2012.

[5] J. B. D. Joshi, E. Bertino, and A. Ghafoor. Temporal hierarchies and inheritance semantics for GTRBAC. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 74–83. ACM, 2002.

[6] E. Medvet, A. Bartoli, B. Carminati, and E. Ferrari. Evolutionary inference of attribute-based access control policies. In *Proceedings of the 8th International Conference on Evolutionary Multi-Criterion Optimization (EMO): Part I*, volume 9018 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2015.

[7] B. Mitra, S. Sural, V. Atluri, and J. Vaidya. Toward mining of temporal roles. In *Proc. 27th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec)*, volume 7964 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2013.

[8] B. Mitra, S. Sural, V. Atluri, and J. Vaidya. The generalized temporal role mining problem. *Journal of Computer Security*, 23(1):31–58, 2015.

[9] B. Mitra, S. Sural, J. Vaidya, and V. Atluri. Mining temporal roles using many-valued concepts. *Computers & Security*, 60:79 – 94, July 2016.

[10] 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):36:1–36:35, 2010.

[11] I. Molloy, Y. Park, and S. Chari. Generative models for access control policies: applications to role mining over logs with attribution. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 45–56. ACM, 2012.

[12] S. D. Stoller and T. Bui. Mining hierarchical temporal roles with multiple metrics. In *Proceedings of the 30th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2016)*, volume 9766 of *Lecture Notes in Computer Science*, pages 79–95. Springer-Verlag, 2016.

[13] E. Uzun, D. Lorenzi, V. Atluri, J. Vaidya, and S. Sural. Migrating from DAC to RBAC. In *Proc. 29th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec)*, volume 9149 of *Lecture Notes in Computer Science*. Springer, 2015.

[14] J. Vaidya, V. Atluri, Q. Guo, and N. Adam. Migrating to optimal RBAC with minimal perturbation. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 11–20. ACM, 2008.

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

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

[17] Z. Xu and S. D. Stoller. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 12(5):533–545, September–October 2015.

[18] D. Zhang, K. Ramamohanarao, and T. Ebringer. Role engineering using graph optimisation. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 139–144, 2007.