Mining Attribute-Based Access Control Policies from Logs

Zhongyuan Xu, Scott D. Stoller

Mining Attribute-Based Access Control Policies from Logs*

Zhongyuan Xu and Scott D. Stoller

Department of Computer Science, Stony Brook University, USA

Abstract. Attribute-based access control (ABAC) provides a high level of flexibility that promotes security and information sharing. ABAC policy mining algorithms have potential to significantly reduce the cost of migration to ABAC, by partially automating the development of an ABAC policy from information about the existing access-control policy and attribute data. This paper presents an algorithm for mining ABAC policies from operation logs and attribute data. To the best of our knowledge, it is the first algorithm for this problem.

1 Introduction

ABAC is becoming increasingly important as security policies become more dynamic and more complex. In industry, more and more products support ABAC, using a standardized ABAC language such as XACML or a vendor-specific ABAC language. In government, the Federal Chief Information Officer Council called out ABAC as a recommended access control model [1, 4]. ABAC allows "an unprecedented amount of flexibility and security while promoting information sharing between diverse and often disparate organizations" [4]. ABAC overcomes some of the problems associated with RBAC, notably role explosion [4].

ABAC promises long-term cost savings through reduced management effort, but manual development of an initial policy can be difficult and expensive [4]. *Policy mining* algorithms promise to drastically reduce the cost of migrating to ABAC, by partially automating the process.

Role mining, i.e., mining of RBAC policies, is an active research area and a currently relatively small (about $70 million) but rapidly growing commercial market segment [3]. In contrast, there is, so far, relatively little work on ABAC policy mining. We recently developed an algorithm to mine an ABAC policy from an ACL policy or RBAC policy [10].

However, an ACL policy or RBAC policy might not be available, e.g., if the current access control policy is encoded in a program or is not enforced by a computerized access control mechanism. An alternative source of information about the current access control policy is operation logs, or "logs" for short. Many software systems produce logs, e.g., for auditing, accounting, and accountability

* This material is based upon work supported in part by NSF under Grant CNS-0831298.

purposes. Molloy, Park, and Chari proposed the idea of mining policies from logs and developed algorithms for mining RBAC policies from logs [6].

The main challenge is that logs generally provide incomplete information about entitlements (i.e., granted permissions). Specifically, logs provide only a lower bound on the entitlements. Therefore, the generated policy should be allowed to include *over-assignments*, i.e., entitlements not reflected in the logs.

This paper presents an algorithm for mining ABAC policies from logs and attribute data. To the best of our knowledge, it is the first algorithm for this problem. It is based on our algorithm for mining ABAC policies from ACLs [10]. At a high level, the algorithm works as follows. It iterates over tuples in the user-permission relation extracted from the log, uses selected tuples as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover additional tuples in the user-permission relation by replacing conjuncts in attribute expressions with constraints. After constructing candidate rules that together cover the entire user-permission relation, it attempts to improve the policy by merging and simplifying candidate rules. Finally, it selects the highest-quality candidate rules for inclusion in the generated policy.

Several changes are needed to our algorithm for mining ABAC policies from ACLs to adapt it to mining from logs. When the algorithm generalizes, merges, or simplifies rules, it discards candidate rules that are invalid, i.e., that produce over-assignments. We modify those parts of the algorithm to consider those candidate rules, because, as discussed above, over-assignments must be permitted. To evaluate those candidate rules, we introduce generalized notions of rule quality and policy quality that quantify a trade-off between the number of over-assignments and other aspects of quality. We consider a metric that includes the normalized number of over-assignments in a weighted sum, a frequency-sensitive variant that assigns higher quality to rules that cover more frequently used entitlements, along the lines of [6], and a metric based on a theory quality metric in inductive logic programming [7, 8].

ABAC policy mining is similar to inductive logic programming (ILP), which learns logic-programming rules from facts. Mining ABAC policies from logs and attribute data is similar to ILP algorithms for learning from positive examples, because those algorithms allow the learned rules to imply more than the given facts (i.e., in our terminology, to have over-assignments). We implemented a translation from ABAC policy mining to Progol [8], a well-known ILP system.

We evaluated our algorithm and the ILP-based approach on some relatively small but non-trivial handwritten case studies and on synthetic ABAC policies. The results demonstrate our algorithm's effectiveness even when the log reflects only a fraction of the entitlements. Although the original (desired) ABAC policy is not reconstructed perfectly from the log, the mined policy is sufficiently similar to it that the mined policy would be very useful as a starting point for policy administrators tasked with developing that ABAC policy.

2 ABAC policy language

This section presents the ABAC policy language used in our work. It is adopted from [10]. We consider a specific ABAC policy language, but our approach is general and can be adapted to other ABAC policy languages. Our ABAC policy language contains the common ABAC policy language constructs, except arithmetic inequalities and negation, which are left for future work.

Given a set U of users and a set A_{u} of user attributes, user attribute data is represented by a function d_{u} such that $d_{\mathrm{u}}(u, a)$ is the value of attribute a for user u. There is a distinguished user attribute uid that has a unique value for each user. Similarly, given a set R of resources and a set A_{r} of resource attributes, resource attribute data is represented by a function d_{r} such that $d_{\mathrm{r}}(r, a)$ is the value of attribute a for resource r. There is a distinguished resource attribute rid that has a unique value for each resource. We assume the set A_{u} of user attributes can be partitioned into a set $A_{\mathrm{u},1}$ of *single-valued user attributes* which have atomic values, and a set $A_{\mathrm{u,m}}$ of *multi-valued user attributes* whose values are sets of atomic values. Similarly, we assume the set A_{r} of resource attributes can be partitioned into a set $A_{\mathrm{r},1}$ of *single-valued resource attributes* and a set of $A_{\mathrm{r,m}}$ of *multi-valued resource attributes*. We assume there is a distinguished atomic value $\perp$ used to indicate that an attribute's value is unknown.

A *user-attribute expression* (UAE) is a function e such that, for each user attribute a, $e(a)$ is either the special value $\top$, indicating that e imposes no constraint on the value of a, or a set (interpreted as a disjunction) of possible values of a excluding $\perp$. We refer to $e(a)$ as the *conjunct* for a. A UAE e *uses* attribute a if $e(a) \neq \top$. Let $\mathrm{attr}(e)$ denote the set of attributes used by e.

A user u *satisfies* a UAE e, denoted $u \models e$, iff $(\forall a \in A_{\mathrm{u},1}.\, e(a) = \top \vee \exists v \in e(a).\, d_{\mathrm{u}}(u, a) = v)$ and $(\forall a \in A_{\mathrm{u,m}}.\, e(a) = \top \vee \exists v \in e(a).\, d_{\mathrm{u}}(u, a) \supseteq v)$. For multi-valued attributes, we use the condition $d_{\mathrm{u}}(u, a) \supseteq v$ instead of $d_{\mathrm{u}}(u, a) = v$ because elements of a multi-valued user attribute typically represent some type of capabilities of a user, so using $\supseteq$ expresses that the user has the specified capabilities and possibly more. For example, suppose $A_{\mathrm{u},1} = \{\mathrm{dept}, \mathrm{position}\}$ and $A_{\mathrm{u,m}} = \{\mathrm{courses}\}$. The function e_1 with $e_1(\mathrm{dept}) = \{\mathrm{CS}\}$ and $e_1(\mathrm{position}) = \{\mathrm{grad}, \mathrm{ugrad}\}$ and $e_1(\mathrm{courses}) = \{\{\mathrm{CS101}, \mathrm{CS102}\}\}$ is a user-attribute expression satisfied by users in the CS department who are either graduate or undergraduate students and whose courses include CS101 and CS102.

In examples, we may write attribute expressions with a logic-based syntax, for readability. For example, the above expression e_1 may be written as $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{ugrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq \{\mathrm{CS101}, \mathrm{CS102}\}$. For an example that uses $\supseteq\in$, the expression e_2 that is the same as e_1 except with $e_2(\mathrm{courses}) = \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$ may be written as $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{ugrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq\in \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$, and is satisfied by graduate or undergraduate students in the CS department whose courses include either CS101 or CS102.

The *meaning* of a user-attribute expression e, denoted $[\![e]\!]_U$, is the set of users in U that satisfy it. User attribute data is an implicit argument to $[\![e]\!]_U$. We say that e *characterizes* the set $[\![e]\!]_U$.

A *resource-attribute expression* (RAE) is defined similarly, except using the set A_{r} of resource attributes instead of the set A_{u} of user attributes. The semantics of RAEs is defined similarly to the semantics of UAEs, except simply using equality, not $\supseteq$, in the condition for multi-valued attributes in the definition of "satisfies", because we do not interpret elements of multi-valued resource attributes specially (e.g., as capabilities).

Constraints express relationships between users and resources. An *atomic constraint* is a formula f of the form $a_{\mathrm{u,m}} \supseteq a_{\mathrm{r,m}}$, $a_{\mathrm{u,m}} \ni a_{\mathrm{r,1}}$, or $a_{\mathrm{u,1}} = a_{\mathrm{r,1}}$, where $a_{\mathrm{u,1}} \in A_{\mathrm{u,1}}$, $a_{\mathrm{u,m}} \in A_{\mathrm{u,m}}$, $a_{\mathrm{r,1}} \in A_{\mathrm{r,1}}$, and $a_{\mathrm{r,m}} \in A_{\mathrm{r,m}}$. The first two forms express that user attributes contain specified values. This is a common type of constraint, because user attributes typically represent some type of capabilities of a user. Let $\mathrm{uAttr}(f)$ and $\mathrm{rAttr}(f)$ refer to the user attribute and resource attribute, respectively, used in f. User u and resource r *satisfy* an atomic constraint f, denoted $\langle u, r\rangle \models f$, if $d_{\mathrm{u}}(u, \mathrm{uAttr}(f)) \neq \bot$ and $d_{\mathrm{r}}(u, \mathrm{rAttr}(f)) \neq \bot$ and formula f holds when the values $d_{\mathrm{u}}(u, \mathrm{uAttr}(f))$ and $d_{\mathrm{r}}(u, \mathrm{rAttr}(f))$ are substituted in it.

A *constraint* is a set (interpreted as a conjunction) of atomic constraints. User u and resource r *satisfy* a constraint c, denoted $\langle u, r\rangle \models c$, if they satisfy every atomic constraint in c. In examples, we write constraints as conjunctions instead of sets. For example, the constraint "specialties $\supseteq$ topics $\wedge$ teams $\ni$ treatingTeam" is satisfied by user u and resource r if the user's specialties include all of the topics associated with the resource, and the set of teams associated with the user contains the treatingTeam associated with the resource.

A *user-permission tuple* is a tuple $\langle u, r, o\rangle$ containing a user, a resource, and an operation. This tuple means that user u has permission to perform operation o on resource r. A *user-permission relation* is a set of such tuples.

A *rule* is a tuple $\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$, where e_{u} is a user-attribute expression, e_{r} is a resource-attribute expression, O is a set of operations, and c is a constraint. For a rule $\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$, let $\mathrm{uae}(\rho) = e_{\mathrm{u}}$, $\mathrm{rae}(\rho) = e_{\mathrm{r}}$, $\mathrm{ops}(\rho) = O$, and $\mathrm{con}(\rho) = c$. For example, the rule $\langle$true, type=task $\wedge$ proprietary=false, {read, request}, projects $\ni$ project $\wedge$ expertise $\supseteq$ expertise$\rangle$ used in our project management case study can be interpreted as "A user working on a project can read and request to work on a non-proprietary task whose required areas of expertise are among his/her areas of expertise." User u, resource r, and operation o *satisfy* a rule ρ, denoted $\langle u, r, o\rangle \models \rho$, if $u \models \mathrm{uae}(\rho) \wedge r \models \mathrm{rae}(\rho) \wedge o \in \mathrm{ops}(\rho) \wedge \langle u, r\rangle \models \mathrm{con}(\rho)$.

An *ABAC policy* is a tuple $\langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, Rules\rangle$, where U, R, A_{u}, A_{r}, d_{u}, and d_{r} are as described above, Op is a set of operations, and $Rules$ is a set of rules.

The user-permission relation induced by a rule ρ is $[\![\rho]\!] = \{\langle u, r, o\rangle \in U \times R \times Op \mid \langle u, r, o\rangle \models \rho\}$. Note that U, R, d_{u}, and d_{r} are implicit arguments to $[\![\rho]\!]$.

The user-permission relation induced by a policy π with the above form is $[\![\pi]\!] = \bigcup_{\rho \in Rules} [\![\rho]\!]$.

3 Problem Definition

An *operation log entry* e is a tuple $\langle u, r, o, t\rangle$ where $u \in U$ is a user, $r \in R$ is a resource, $o \in Op$ is an operation, and t is a timestamp. An *operation log* is a sequence of operation log entries. The user-permission relation induced by an operation log L is $UP(L) = \{\langle u, r, o\rangle \mid \exists t.\ \langle u, r, o, t\rangle \in L\}$.

The input to the *ABAC-from-logs policy mining problem* is a tuple $I = \langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, L\rangle$, where U is a set of users, R is a set of resources, Op is a set of operations, A_{u} is a set of user attributes, A_{r} is a set of resource attributes, d_{u} is user attribute data, d_{r} is resource attribute data, and L is an operation log, such that the users, resources, and operations that appear in L are subsets of U, R, and Op, respectively. The goal of the problem is to find a set of rules *Rules* such that the ABAC policy $\pi = \langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, Rules\rangle$ maximizes a suitable policy quality metric.

The policy quality metric should reflect the size and meaning of the policy. Size is measured by *weighted structural complexity* (WSC) [5], and smaller policies are considered to have higher quality. This is consistent with usability studies of access control rules, which conclude that more concise policies are more manageable. Informally, the WSC of an ABAC policy is a weighted sum of the number of elements in the policy. Specifically, the WSC of an attribute expression is the number of atomic values that appear in it, the WSC of an operation set is the number of operations in it, the WSC of a constraint is the number of atomic constraints in it, and the WSC of a rule is a weighted sum of the WSCs of its components, namely, $\mathrm{WSC}(\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle) = w_1\mathrm{WSC}(e_{\mathrm{u}}) + w_2\mathrm{WSC}(e_{\mathrm{r}}) + w_3\mathrm{WSC}(O) + w_4\mathrm{WSC}(c)$, where the w_i are user-specified weights. The WSC of a set of rules is the sum of the WSCs of its members.

The meaning $[\![\pi]\!]$ of the ABAC policy is taken into account by considering the differences from $UP(L)$, which consist of over-assignments and under-assignments. The over-assignments are $[\![\pi]\!] \setminus UP(L)$. The under-assignments are $UP(L) \setminus [\![\pi]\!]$. Since logs provide only a lower-bound on the actual user-permission relation (a.k.a entitlements), it is necessary to allow some over-assignments, but not too many. Allowing under-assignments is beneficial if the logs might contain noise, in the form of log entries representing uses of permissions that should not be granted, because it reduces the amount of such noise that gets propagated into the policy, and it improves the stability of the generated policy. We define a policy quality metric that is a weighted sum of these aspects:

$$Q_{\mathrm{pol}}(\pi, L) = \mathrm{WSC}(\pi) + w_o \,|\, [\![\pi]\!] \setminus UP(L)| \,/\, |U| \tag{1}$$

where the *policy over-assignment weight* w_o is a user-specified weight for over-assignments, and for a set S of user-permission tuples, the frequency-weighted size of S with respect to log L is $|S|_L = \sum_{\langle u,r,o\rangle \in S} \mathrm{freq}(\langle u, r, o\rangle, L)$, where the relative frequency of a user-permission tuple in a log is given by the *frequency function* $\mathrm{freq}(\langle u, r, o\rangle, L) = |\{e \in L \mid \mathrm{userPerm}(e) = \langle u, r, o\rangle\}| \,/\, |L|$, where the user-permission part of a log entry is given by $\mathrm{userPerm}(\langle u, r, o, t\rangle) = \langle u, r, o\rangle$.

For simplicity, our presentation of the problem and algorithm assume that attribute data does not change during the time covered by the log. Accommo-

dating changes to attribute data is not difficult. It mainly requires re-defining the notions of policy quality and rule quality (introduced in Section 4) to be based on the set of log entries covered by a rule, denoted $[\![\rho]\!]_{\mathrm{LE}}$, rather than $[\![\rho]\!]$. The definition of $[\![\rho]\!]_{\mathrm{LE}}$ is similar to the definition of $[\![\rho]\!]$, except that, when determining whether a log entry is in $[\![\rho]\!]_{\mathrm{LE}}$, the attribute data in effect at the time of the log entry is used.

4 Algorithm

Our algorithm is based on the algorithm for mining ABAC policies from ACLs and attribute data in [10]. Our algorithm does not take the order of log entries into account, so the log can be summarized by the user-permission relation UP_0 induced by the log and the frequency function freq, described in the penultimate paragraph of Section 3.

Top-level pseudocode appears in Figure 1. We refer to tuples selected in the first statement of the first while loop as *seeds*. The top-level pseudocode is explained by embedded comments. It calls several functions, described next. Function names hyperlink to pseudocode for the function, if it is included in the paper, otherwise to the description of the function.

The function addCandRule($s_{\mathrm{u}}, s_r, s_{\mathrm{o}}, cc, uncovUP, Rules$) in Figure 2 first calls computeUAE to compute a user-attribute expression e_{u} that characterizes s_{u}, and computeRAE to compute a resource-attribute expression e_{r} that characterizes s_r. It then calls generalizeRule($\rho, cc, uncovUP, Rules$) to generalize rule $\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, s_{\mathrm{o}}, \emptyset \rangle$ to ρ' and adds ρ' to candidate rule set *Rules*. The details of the functions called by addCandRule are described next.

The function computeUAE(s, U) computes a user-attribute expression e_{u} that characterizes the set s of users. Preference is given to attribute expressions that do not use uid, since attribute-based policies are generally preferable to identity-based policies, even when they have higher WSC, because attribute-based generalize better. Similarly, computeRAE(s, R) computes a resource-attribute expression that characterizes the set s of resources. Pseudocode for computeUAE and computeRAE are omitted. The function candidateConstraint(r, u) returns a set containing all of the atomic constraints that hold between resource r and user u. Pseudocode for candidateConstraint is straightforward and omitted.

The function generalizeRule($\rho, cc, uncovUP, Rules$) in Figure 3 attempts to generalize rule ρ by adding some of the atomic constraints in cc to ρ and eliminating the conjuncts of the user attribute expression and/or the resource attribute expression corresponding to the attributes used in those constraints, i.e., mapping those attributes to $\top$. We call a rule obtained in this way a *generalization* of ρ. Such a rule is more general than ρ in the sense that it refers to relationships instead of specific values. Also, the user-permission relation induced by a generalization of ρ is a superset of the user-permission relation induced by ρ. generalizeRule($\rho, cc, uncovUP, Rules$) returns the generalization ρ' of ρ with the best quality according to a given rule quality metric. Note that ρ' may cover

tuples that are already covered (i.e., are in UP); in other words, our algorithm can generate policies containing rules whose meanings overlap.

A *rule quality metric* is a function $Q_{\text{rul}}(\rho, UP)$ that maps a rule ρ to a totally-ordered set, with the ordering chosen so that larger values indicate high quality. The second argument UP is a set of user-permission tuples. Our rule quality metric assigns higher quality to rules that cover more currently uncovered user-permission tuples and have smaller size, with an additional term that imposes a penalty for over-assignments, measured as a fraction of the number of user-permission tuples covered by the rule, and with a weight specified by a parameter w'_o, called the *rule over-assignment weight*.

$$Q_{\text{rul}}(\rho, UP) = \frac{|\,[\![\rho]\!] \cap UP|}{|\rho|} \times (1 - \frac{w'_o \times |\text{overAssign}(\rho)|}{|\,[\![\rho]\!]\,|}).$$

In generalizeRule, *uncovUP* is the second argument to Q_{rul}, so $[\![\rho]\!] \cap UP$ is the set of user-permission tuples in UP_0 that are covered by ρ and not covered by rules already in the policy. The loop over i near the end of the pseudocode for generalizeRule considers all possibilities for the first atomic constraint in cc that gets added to the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in c that get added to the constraint.

We also developed a frequency-sensitive variant of this rule quality metric. Let $Q^{\text{freq}}_{\text{rul}}$ denote the frequency-weighted variant of Q_{rul}, obtained by weighting each user-permission tuple by its relative frequency (i.e., fraction of occurrences) in the log, similar to the definition of λ-distance in [6]. Specifically, the definition of $Q^{\text{freq}}_{\text{rul}}$ is obtained from the definition of Q_{rul} by replacing $|\,[\![\rho]\!] \cap UP|$ with $|\,[\![\rho]\!] \cap UP|_L$ (recall that $|\cdot|_L$ is defined in Section 3).

We also developed a rule quality metric $Q^{\text{ILP}}_{\text{rul}}$ based closely on the theory quality metric for inductive logic programming described in [7]. Details of the definition are omitted to save space.

The function mergeRules(*Rules*) in Figure 3 attempts to improve the quality of *Rules* by removing redundant rules and merging pairs of rules. A rule ρ in *Rules* is *redundant* if *Rules* contains another rule ρ' such that every user-permission tuple in UP_0 that is in $[\![\rho]\!]$ is also in $[\![\rho']\!]$. Informally, rules ρ_1 and ρ_2 are merged by taking, for each attribute, the union of the conjuncts in ρ_1 and ρ_2 for that attribute. If adding the resulting rule ρ_{mrg} to the policy and removing rules (including ρ_1 and ρ_2) that become redundant improves policy quality and does not introduce over-assignments where none existed before, then ρ_{mrg} is added to *Rules*, and the redundant rules are removed from *Rules*. As optimizations (in the implementation, not reflected in the pseudocode), meanings of rules are cached, and policy quality is computed incrementally. mergeRules(*Rules*) updates its argument *Rules* in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(*Rules*) attempts to simplify all of the rules in *Rules*. It updates its argument *Rules* in place, replacing rules in *Rules* with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several

```
// Rules is the set of candidate rules
Rules = ∅
// uncovUP contains user-permission tuples
// in UP_0 that are not covered by Rules
uncovUP = UP_0.copy()
while ¬uncovUP.isEmpty()
  // Select an uncovered tuple as a "seed".
  ⟨u, r, o⟩ = some tuple in uncovUP
  cc = candidateConstraint(r, u)
  // s_u contains users with permission ⟨r, o⟩
  // and that have the same candidate
  // constraint for r as u
  s_u = {u' ∈ U | ⟨u', r, o⟩ ∈ UP_0
            ∧ candidateConstraint(r, u') = cc}
  addCandRule(s_u, {r}, {o}, cc, uncovUP, Rules)
  // s_o is set of operations that u can apply to r
  s_o = {o' ∈ Op | ⟨u, r, o'⟩ ∈ UP_0}
  addCandRule({u}, {r}, s_o, cc, uncovUP, Rules)
end while
```

```
// Repeatedly merge and simplify
// rules, until this has no effect
mergeRules(Rules)
while simplifyRules(Rules)
        && mergeRules(Rules)
  skip
end while
// Select high quality rules into Rules'.
Rules' = ∅
Repeatedly move highest-quality rule
from Rules to Rules' until
∑_{ρ∈Rules'} [[ρ]] ⊇ UP_0, using
UP_0 \ [[Rules']] as second argument to
Q_rul, and discarding a rule if it does
not cover any tuples in UP_0 currently
uncovered by Rules'.
return Rules'
```

Fig. 1. Policy mining algorithm. The pseudocode starts in column 1 and continues in column 2.

```
function addCandRule(s_u, s_r, s_o, cc, uncovUP, Rules)
// Construct a rule ρ that covers user-perm. tuples {⟨u, r, o⟩ | u ∈ s_u ∧ r ∈ s_r ∧ o ∈ s_o}.
e_u = computeUAE(s_u, U);  e_r = computeRAE(s_r, R);  ρ = ⟨e_u, e_r, s_o, ∅⟩
ρ' = generalizeRule(ρ, cc, uncovUP, Rules);  Rules.add(ρ');  uncovUP.removeAll([[ρ']])
```

Fig. 2. Compute a candidate rule ρ' and add ρ' to candidate rule set *Rules*

ways, including elimination of redundant sets using function elimRedundantSets, elimination of conjuncts, elimination of constraints, elimination of elements of sets in conjuncts for multi-valued user attributes, and elimination of overlap between rules. The detailed definition is similar to the one in [10] and is omitted to save space.

4.1 Example

We illustrate the algorithm on a small fragment of our university case study (*cf.* Section 5.1). The fragment contains a single rule $\rho_0 = \langle \text{true}, \textit{type} \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{crsTaught} \ni \text{crs}\rangle$ and all of the attribute data from the full case study, except attribute data for gradebooks for courses other than cs601. We consider an operation log L containing three entries: $\{\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore}, \text{t}_1\rangle, \langle \text{csFac2}, \text{cs601gradebook}, \text{readScore}, \text{t}_2\rangle, \langle \text{csStu3}, \text{cs601gradebook}, \text{addScore}, \text{t}_3\rangle\}$. User csFac2 is a faculty in the computer science department who is teaching cs601; attributes are position = faculty, dept = cs, and crsTaught = {cs601}. csStu3 is a CS student who is a TA of cs601; attributes are position =

```
function generalizeRule(ρ, cc, uncovUP,
                        Rules)
// ρ_best is best generalization of ρ
ρ_best = ρ
// gen[i][j] is a generalization of ρ using
// cc'[i]
gen = new Rule[cc.length][3]
for i = 1 to cc.length
  f = cc[i]
  // generalize by adding f and eliminating
  // conjuncts for both attributes used in f.
  gen[i][1] = ⟨uae(ρ)[uAttr(f) ↦ ⊤],
              rae(ρ)[rAttr(f) ↦ ⊤],
              ops(ρ), con(ρ) ∪ {f}⟩
  // generalize by adding f and eliminating
  // conjunct for user attribute used in f
  gen[i][2] = ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ),
              ops(ρ), con(ρ) ∪ {f}⟩
  // generalize by adding f and eliminating
  // conjunct for resource attrib. used in f.
  gen[i][3] = ⟨uae(ρ), rae(ρ)[rAttr(f) ↦ ⊤],
              ops(ρ), con(ρ) ∪ {f}⟩
end for
for i = 1 to cc.length and j = 1 to 3
  // try to further generalize gen[i]
  ρ'' = generalizeRule(gen[i][j], cc[i+1 ..],
                       uncovUP, Rules)
  if Q_rul(ρ'', uncovUP) > Q_rul(ρ_best,
                                 uncovUP)
    ρ_best = ρ''
  end if
end for
return ρ_best
```

```
function mergeRules(Rules)
// Remove redundant rules
redun = {ρ ∈ Rules | ∃ ρ' ∈ Rules \ {ρ}.
           [[ρ]] ∩ UP_0 ⊆ [[ρ']] ∩ UP_0}
Rules.removeAll(redun)
// Merge rules
workSet = {(ρ_1, ρ_2) | ρ_1 ∈ Rules ∧ ρ_2 ∈ Rules
            ∧ ρ_1 ≠ ρ_2 ∧ con(ρ_1) = con(ρ_2)}
while not(workSet.empty())
  (ρ_1, ρ_2) = workSet.remove()
  ρ_mrg = ⟨uae(ρ_1) ∪ uae(ρ_2),
           rae(ρ_1) ∪ rae(ρ_2),
           ops(ρ_1) ∪ ops(ρ_2), con(ρ_1)⟩
  // Find rules that become redundant
  // if merged rule ρ_mrg is added
  redun = {ρ ∈ Rules | [[ρ]] ⊆ [[ρ_mrg]]}
  // Add the merged rule and remove redun-
  // dant rules if this improves policy quality
  // and does not introduce over-assignments.
  // where none existed before.
  if Q_pol(Rules ∪ {ρ_mrg} \ redun) < Q_pol(Rules)
     ∧ (noOA(ρ_1) ∧ noOA(ρ_2) ⇒ noOA(ρ_mrg))
    Rules.removeAll(redun)
    workSet.removeAll({(ρ_1, ρ_2) ∈ workSet |
          ρ_1 ∈ redun ∨ ρ_2 ∈ redun})
    workSet.addAll({(ρ_mrg, ρ) | ρ ∈ Rules
                    ∧ con(ρ) = con(ρ_mrg)})
    Rules.add(ρ_mrg)
  end if
end while
return true if any rules were merged
```

Fig. 3. Left: Generalize rule ρ by adding some formulas from *cc* to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i. Right: Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$. $\text{noOA}(\rho)$ holds if ρ has no over-assignments, i.e., $[\![\rho]\!] \subseteq UP_0$.

student, dept = cs, and crsTaught = {cs601}. cs601gradebook is a resource with attributes type = gradebook, dept = cs, and crs = cs601.

Our algorithm selects user-permission tuple ⟨csFac2, cs601gradebook, addScore⟩ as the first seed, and calls function candidateConstraint to compute the set of atomic constraints that hold between csFac2 and cs601gradebook; the result is

$cc = \{\text{dept} = \text{dept}, \text{crsTaught} \ni \text{crs}\}$. addCandRule is called twice to compute candidate rules. The first call to addCandRule calls computeUAE to compute a UAE e_{u} that characterizes the set s_{u} containing users with permission $\langle \text{addScore}, \text{cs601gradebook} \rangle$ and with the same candidate constraint as csFac2 for cs601gradebook; the result is $e_{\text{u}} = \{\text{position} \in \{\text{faculty}, \text{student}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{crsTaught} \supseteq \{\{\text{cs601}\}\}\}$. addCandRule also calls computeRAE to compute a resource-attribute expression that characterizes {cs601gradebook}; the result is $e_{\text{r}} = \{\text{crs} \in \{\text{cs601}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{type} \in \{\text{gradebook}\}\}$. The set of operations considered in this call to addCandRule is simply $s_{\text{o}} = \{\text{addScore}\}$. addCandRule then calls generalizeRule, which generates a candidate rule ρ_1 which initially has e_{u}, e_{r} and s_{o} in the first three components, and then atomic constraints in cc are added to ρ_1, and conjuncts in e_{u} and e_{r} for attributes used in cc are eliminated; the result is $\rho_1 = \langle \text{position} \in \{\text{faculty}, \text{student}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$, which also covers the third log entry. Similarly, the second call to addCandRule generates a candidate rule $\rho_2 = \langle \text{position} \in \{\text{faculty}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$, which also covers the second log entry.

All of $UP(L)$ is covered, so our algorithm calls mergeRules, which attempts to merge ρ_1 and ρ_2 into rule $\rho_3 = \langle \text{position} \in \{\text{faculty}, \text{student}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore, readScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$. ρ_3 is discarded because it introduces an over-assignment while ρ_1 and ρ_2 do not. Next, simplifyRules is called, which first simplifies ρ_1 and ρ_2 to ρ_1' and ρ_2', respectively, and then eliminates ρ_1' because it covers a subset of the tuples covered by ρ_2'. The final result is ρ_2', which is identical to the rule ρ_0 in the original policy.

5 Evaluation Methodology

We evaluate our policy mining algorithms on synthetic operation logs generated from an ABAC policy (some handwritten and some synthetic) and probability distributions characterizing the frequency of actions. This allows us to evaluate the effectiveness of our algorithm by comparing the mined policies with the original ABAC policies. We are eager to also evaluate our algorithm on actual operation logs and actual attribute data, when we are able to obtain them.

5.1 ABAC Policies

Case Studies. We developed four case studies for use in evaluation of our algorithm, described briefly here. Details of the case studies, including all policy rules, various size metrics (number of users, number of resources, etc.), and some illustrative attribute data, appear in [10].

Our *university case study* is a policy that controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. Our *health care case study* is a policy that controls access by nurses, doctors, patients, and agents (e.g., a patient's spouse) to electronic health records

(HRs) and HR items (i.e., entries in health records). Our *project management case study* is a policy that controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. Our *online video case study* is a policy that controls access to videos by users of an online video service.

The number of rules in the case studies is relatively small (10 ± 1 for the first three case studies, and 6 for online video), but they express non-trivial policies and exercise all the features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The manually written attribute dataset for each case study contains a small number of instances of each type of user and resource.

For the first three case studies, we generated a series of synthetic attribute datasets, parameterized by a number N, which is the number of departments for the university and project management case studies, and the number of wards for the health care case study. The generated attribute data for users and resources associated with each department or ward are similar to but more numerous than the attribute data in the manually written datasets. We did not bother creating synthetic data for the online video case study, because the rules are simpler.

Synthetic Policies. We generated synthetic policies using the algorithm proposed by Xu and Stoller [10]. Briefly, the policy synthesis algorithm first generates the rules and then uses the rules to guide generation of the attribute data; this allows control of the number of granted permissions. The algorithm takes N_{rule}, the desired number of rules, as an input. The numbers of users and resources are proportional to the number of rules. Generation of rules and attribute data is based on several probability distributions, which are based loosely on the case studies or assumed to have a simple functional form (e.g., uniform distribution).

5.2 Log Generation

The inputs to the algorithm are an ABAC policy π, the desired completeness of the log, and several probability distributions. The *completeness* of a log, relative to an ABAC policy, is the fraction of user-permission tuples in the meaning of the policy that appear in at least one entry in the log. A straightforward log generation algorithm would generate each log entry by first selecting an ABAC rule, according to a probability distribution on rules, and then selecting a user-permission tuple that satisfies the rule, according to probability distributions on users, resources, and operations. This process would be repeated until the specified completeness is reached. This algorithm is inefficient when high completeness is desired. Therefore, we adopt a different approach that takes advantage of the fact that our policy mining algorithm is insensitive to the order of log entries and depends only on the frequency of each user-permission tuple in the log. In particular, instead of generating logs (which would contain many entries for popular user-permission tuples), our algorithm directly generates a *log summary*, which is a set of user-permission tuples with associated frequencies (equivalently, a set of user-permission tuples and a frequency function).

Probability Distributions. An important characteristic of the probability distributions used in synthetic log and log summary generation is the ratio between the most frequent (i.e., most likely) and least frequent items of each type (rule, user, etc.). For case studies with manually written attribute data, we manually created probability distributions in which this ratio ranges from about 3 to 6. For case studies with synthetic data and synthetic policies, we generated probability distributions in which this ratio is 25 for rules, 25 for resources, 3 for users, and 3 for operations (the ratio for operations has little impact, because it is relevant only when multiple operations appear in the same rule, which is uncommon).

5.3 Metrics

For each case study and each associated attribute dataset (manually written or synthetic), we generate a synthetic operation log using the algorithm in Section 5.2 and then run our ABAC policy mining algorithms. We evaluate the effectiveness of each algorithm by comparing the generated ABAC policy to the original ABAC policy, using the metrics described below.

Syntactic Similarity. Jaccard similarity of sets is $J(S_1, S_2) = |S_1 \cap S_2| \,/\, |S_1 \cup S_2|$. Syntactic similarity of UAEs is defined by $S^{\mathrm{u}}_{\mathrm{syn}}(e, e') = |A_{\mathrm{u}}|^{-1} \sum_{a \in A_{\mathrm{u}}} J(e(a), e'(a))$. Syntactic similarity of RAEs is defined by $S^{\mathrm{r}}_{\mathrm{syn}}(e, e') = |A_{\mathrm{r}}|^{-1} \sum_{a \in A_{\mathrm{r}}} J(e(a), e'(a))$. The syntactic similarity of rules $\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$ and $\langle e'_{\mathrm{u}}, e'_{\mathrm{r}}, O', c'\rangle$ is the average of the similarities of their components, specifically, the average of $S^{\mathrm{u}}_{\mathrm{syn}}(e_{\mathrm{u}}, e'_{\mathrm{u}})$, $S^{\mathrm{r}}_{\mathrm{syn}}(e_{\mathrm{r}}, e'_{\mathrm{r}})$, $J(O, O')$, and $J(c, c')$. The *syntactic similarity* of rule sets *Rules* and *Rules*$'$ is the average, over rules ρ in *Rules*, of the syntactic similarity between ρ and the most similar rule in *Rules*$'$. The *syntactic similarity* of policies π and π' is the maximum of the syntactic similarities of the sets of rules in the policies, considered in both orders (this makes the relation symmetric). Syntactic similarity ranges from 0 (completely different) to 1 (identical).

Semantic Similarity. Semantic similarity measures the similarity of the entitlements granted by two policies. The *semantic similarity* of policies π and π' is defined by $J(\llbracket\pi\rrbracket, \llbracket\pi'\rrbracket)$. Semantic similarity ranges from 0 (completely different) to 1 (identical).

Fractions of Under-Assignments and Over-Assignments. To characterize the semantic differences between an original ABAC policy π_0 and a mined policy π in a way that distinguishes under-assignments and over-assignments, we compute the fraction of over-assignments and the fraction of under-assignments, defined by $|\llbracket\pi\rrbracket \setminus \llbracket\pi_0\rrbracket| \,/\, |\llbracket\pi\rrbracket|$ and $|\llbracket\pi_0\rrbracket \setminus \llbracket\pi\rrbracket| \,/\, |\llbracket\pi\rrbracket|$, respectively.

6 Experimental Results

This section presents experimental results using an implementation of our algorithm in Java. The implementation, case studies, and synthetic policies used in the experiments are available at `http://www.cs.stonybrook.edu/~stoller/`.

Over-Assignment Weight. The optimal choice for the over-assignment weights w_o and w'_o in the policy quality and rule quality metrics, respectively, depends on the log completeness. When log completeness is higher, fewer over-assignments are desired, and larger over-assignments weights give better results. In experiments, we take $w_o = 50c - 15$ and $w'_o = w_o/10$, where c is log completeness. In a production setting, the exact log completeness would be unknown, but a rough estimate suffices, because our algorithm's results are robust to error in this estimate. For example, for case studies with manually written attribute data, when the actual log completeness is 80%, and the estimated completeness used to compute w_o varies from 70% to 90%, the semantic similarity of the original and mined policies varies by 0.04, 0.02, and 0 for university, healthcare, and project management, respectively.

Experimental Results. Figure 4 shows results from our algorithm. In each graph, curves are shown for the university, healthcare, and project management case studies with synthetic attribute data with N equal to 6, 10, and 10, respectively (average over results for 10 synthetic datasets, with 1 synthetic log per synthetic dataset), the online video case study with manually written attribute data (average over results for 10 synthetic logs), and synthetic policies with $N_{\text{rule}} = 20$ (average over results for 10 synthetic policies, with 1 synthetic log per policy). Error bars show standard deviation. Running time is at most 12 sec for each problem instance in our experiments.

For log completeness 100%, all four case study policies are reconstructed exactly, and the semantics of synthetic policies is reconstructed almost exactly: the semantic similarity is 0.98. This is a non-trivial result, especially for the case studies: an algorithm could easily generate a policy with over-assignments or generate more complex rules. As expected, the results get worse as log completeness decreases. When evaluating the results, it is important to consider what levels of log completeness are likely to be encountered in practice. One datapoint comes from Molloy *et al.*'s work on role mining from real logs [6]. For the experiments in [6, Tables 4 and 6], the actual policy is not known, but their algorithm produces policies with 0.52% or fewer over-assignments relative to $UP(L)$, and they interpret this as a good result, suggesting that they consider the log completeness to be near 99%. Based on this, we consider our experiments with log completeness below 90% to be severe stress tests, and results for log completeness 90% and higher to be more representative of typical results in practice.

Syntactic similarity for all four case studies is above 0.87 for log completeness 60% or higher, and is above 0.93 for log completeness 80% or higher. Syntactic similarity is lower for synthetic policies, but this is actually a good result. The synthetic policies tend to be unnecessarily complicated, and the mined policies are better in the sense that they have lower WSC. For example, for 100% log completeness, the mined policies have 0.98 semantic similarity to the synthetic policies (i.e., the meaning is almost the same), but the mined policies are simpler, with WSC 17% less than the original synthetic policies.

Semantic similarity is above 0.7 for log completeness 60% or higher, and above 0.89 for log completeness 80% or higher, for synthetic policies and for case

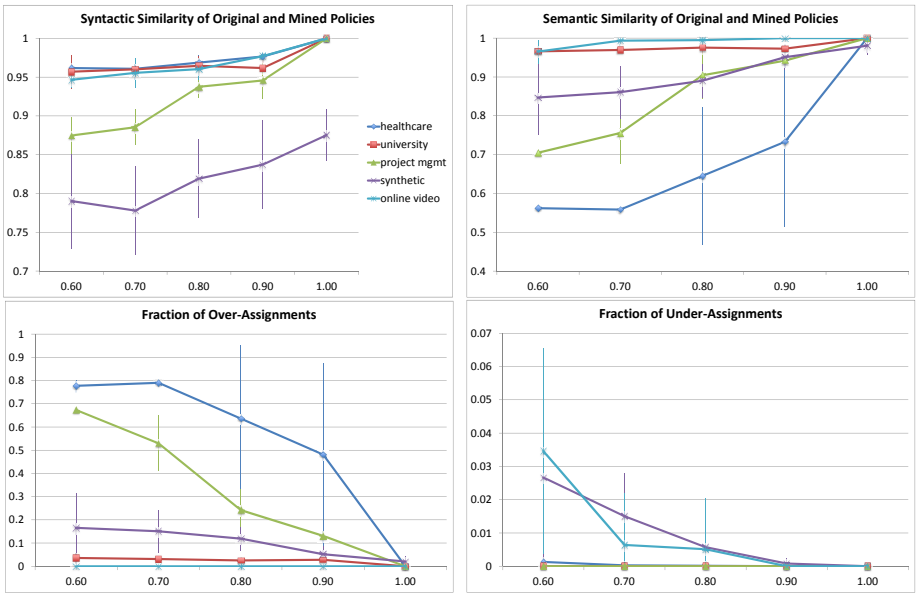

Fig. 4. Top: Syntactic similarity and semantic similarity of original and mined ABAC policies, as a function of log completeness. Bottom: Fractions of over-assignments and under-assignments in mined ABAC policy, as a function of log completeness. The legend (omitted from some graphs to save space) is the same for all four graphs.

studies other than healthcare. The semantic similarity is lower for healthcare, because the over-assignment weight given by the above formula is not optimal for this policy. In fact, if the optimal value of w_o is used for each log completeness, the semantic similarity for healthcare is always above 0.99. Better automated tuning of w_o is a direction for future work.

The fractions of over-assignments and under-assignments are below 0.24 and 0.05, respectively, when log completeness is 80% or higher, for synthetic policies and for case studies other than healthcare. The fractions are higher for healthcare, because w_o is not well chosen, as discussed above.

Comparison of Rule Quality Metrics. The above experiments use the first rule quality metric, Q_{rul}, in Section 4. We also performed experiments using $Q_{\text{rul}}^{\text{freq}}$ and $Q_{\text{rul}}^{\text{ILP}}$ on case studies with manually written attribute data and synthetic policies. We found that there is no clear winner between Q_{rul} and $Q_{\text{rul}}^{\text{freq}}$ (sometimes one is better, sometimes the other is better), and $Q_{\text{rul}}^{\text{ILP}}$ gives worse results overall.

Comparison with Inductive Logic Programming. To translate ABAC policy mining from logs to Progol [8], we used the translation of ABAC policy mining from ACLs to Progol in [10, Sections 5.5, 16], except negative examples corresponding to absent user-permission tuples are omitted from the generated program, and the statement `set(posonly)?` is included, telling Progol to use its algorithm for

learning from positive examples. For the four case studies with manually written attribute data (in contrast, Figure 4 uses synthetic attribute for three of the case studies), for log completeness 100%, semantic similarity of the original and Progol-mined policies ranges from 0.37 for project management and healthcare to 0.93 for online video, while our algorithm exactly reconstructs all four policies.

7 Related Work

We are not aware of prior work on ABAC mining from logs. The closest topics of related work are ABAC mining from ACLs and role mining from logs.

7.1 ABAC Policy Mining from ACLs

Our policy mining algorithm is based on our algorithm for ABAC policy mining from ACLs [10]. The main differences are described in Section 1.

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [9]. In the most closely related part of their work, a supervised machine learning algorithm is used to learn classifiers (analogous to attribute expressions) that associate users with roles, given as input the users, the roles, user attribute data, and the user-role assignment. Perhaps the largest difference between their work and ABAC policy mining is that their approach needs to be given the roles and the role-permission or user-role assignment as training data; in contrast, ABAC policy mining algorithms do not require any part of the desired high-level policy to be given as input. Also, their work does not consider anything analogous to constraints and does not attempt to optimize the size or readability of the generated policy.

Association rule mining is another possible basis for ABAC policy mining. However, association rule mining algorithms are not well suited to ABAC policy mining, because they 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 that are strictly satisfied, that completely cover the input data, and are minimum-sized among such sets of rules. Consequently, unlike our algorithm, they do not give preference to smaller rules or rules with less overlap.

7.2 Role Mining from Logs

Gal-Oz *et al.* [2] assume that logs record sets of permissions exercised together in one high-level operation. Their role mining algorithm introduces roles whose sets of assigned permissions are the sets of permissions in the log. Their algorithm introduces over-assignments by removing roles with few users or whose permission set occurs few times in the log and re-assigning their members to roles with more permissions. Their algorithm does not use attribute data.

Molloy *et al.* apply a machine learning algorithm that uses a statistical approach, based upon a generative model, to find the policy that is most likely to generate the behavior (usage of permissions) observed in the logs [6]. They

give an algorithm, based on Rosen-Zvi et al.'s algorithm for learning Author-Topic Models (ATMs), to mine meaningful roles from logs and attribute data, i.e., roles such that the user-role assignment is statistically correlated with user attributes. Their approach can be adapted to ABAC policy mining from logs, but its scalability in this context is questionable, because the adapted algorithm would enumerate and then rank all tuples containing a UAE, RAE and constraint (i.e., all tuples with the components of a candidate rule other than the operation set), and the number of such tuples is very large. In contrast, our algorithm never enumerates such candidates.

Zhang *et al.* use machine learning algorithms to improve the quality of a given role hierarchy based on users' access patterns as reflected in operation logs [12, 11]. These papers do not consider improvement or mining of ABAC policies.

References

1. Federal Chief Information Officer Council: Federal Identity Credential and Access Management (FICAM) Roadmap and Implementation Guidance, ver. 2.0 (2011)
2. Gal-Oz, N., Gonen, Y., Yahalom, R., Gudes, E., Rozenberg, B., Shmueli, E.: Mining roles from web application usage patterns. In: Proc. 8th Int'l. Conference on Trust, Privacy and Security in Digital Business (TrustBus). pp. 125–137. Springer (2011)
3. Hachana, S., Cuppens-Boulahia, N., Cuppens, F.: Role mining to assist authorization governance: How far have we gone? International Journal of Secure Software Engineering 3(4), 45–64 (October-December 2012)
4. Hu, V.C., Ferraiolo, D., Kuhn, R., Friedman, A.R., Lang, A.J., Cogdell, M.M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K.: Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Final Draft). NIST Special Publication 800-162, National Institute of Standards and Technology (Sep 2013)
5. Molloy, I., Chen, H., Li, T., Wang, Q., Li, N., Bertino, E., Calo, S.B., Lobo, J.: Mining roles with multiple objectives. ACM Trans. Inf. Syst. Secur. 13(4) (2010)
6. Molloy, I., Park, Y., Chari, S.: 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). ACM (2012)
7. Muggleton, S.H.: Inverse entailment and progol. New Generation Computing 13, 245–286 (1995)
8. Muggleton, S.H., Firth, J.: CProgol4.4: a tutorial introduction. In: Dzeroski, S., Lavrac, N. (eds.) Relational Data Mining, pp. 160–188. Springer-Verlag (2001)
9. Ni, Q., Lobo, J., Calo, S., Rohatgi, P., Bertino, E.: Automating role-based provisioning by learning from examples. In: Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 75–84. ACM (2009)
10. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies. Computing Research Repository (CoRR) abs/1306.2401 (Jun 2013), revised January 2014. `http://arxiv.org/abs/1306.2401`.
11. Zhang, W., Chen, Y., Gunter, C.A., Liebovitz, D., Malin, B.: Evolving role definitions through permission invocation patterns. In: Proc. 18th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 37–48. ACM (2013)
12. Zhang, W., Gunter, C.A., Liebovitz, D., Tian, J., Malin, B.: Role prediction using electronic medical record system audits. In: AMIA Annual Symposium Proceedings. pp. 858–867. American Medical Informatics Association (2011)