

Mining Attribute-Based Access Control Policies from Logs*

Zhongyuan Xu and Scott D. Stoller

Department of Computer Science, Stony Brook University, USA

July 28, 2014

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 [37] or a vendor-specific ABAC language. In government, the Federal Chief Information Officer Council called out ABAC as a recommended access control model [6, 14]. ABAC allows “an unprecedented amount of flexibility and security while promoting information sharing between diverse and often disparate organizations” [14]. ABAC overcomes some of the problems associated with RBAC, notably role explosion [14], which makes RBAC policies large and hard to manage.

ABAC promises long-term cost savings through reduced management effort, but manual development of an initial policy can be difficult [2] and expensive [14]. *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 (e.g., [15, 30, 34, 32, 41, 3, 7, 12, 16, 33, 4, 21, 8, 18, 20, 22, 31, 35, 5, 17, 23, 36, 38, 9, 19, 40, 42]) and a currently relatively small (about \$70 million) but rapidly growing commercial market segment [13]. Some form of role mining is supported by security products from companies of all sizes, from IT giants (e.g., IBM Tivoli Access Manager, Oracle Role Manager, CA Technologies GovernanceMinder) to small start-ups (e.g., Bay31 Role Designer). 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 [39]; to the best of our knowledge, it is the first algorithm for that problem.

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 purposes. Molloy, Park, and Chari proposed the idea of mining policies from logs and developed algorithms for mining RBAC policies from logs [23].

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

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 [39]. 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 [23], and a metric based on a theory quality metric in inductive logic programming [26, 24, 27].

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 [24, 27], 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.

2 ABAC policy language

This section presents the ABAC policy language used in our work. It is adopted from [39]. For simplicity and concreteness, 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.

ABAC policies refer to attributes of users and resources. 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 $d_u(u, a)$ 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 $d_r(r, a)$ 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 $A_{u,1}$ of *single-valued user attributes* which have atomic values, and a set $A_{u,m}$ 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 $A_{r,1}$ of *single-valued resource attributes* and a set of $A_{r,m}$ of *multi-valued resource attributes*. Let Val_s be the set of possible atomic values of attributes. We assume Val_s includes a distinguished value \perp used to indicate that an attribute’s value is unknown. The set of possible values of multi-valued attributes

is $\text{Val}_m = \text{Set}(\text{Val}_s \setminus \{\perp\}) \cup \perp$, where $\text{Set}(S)$ is the powerset of set S .

Attribute expressions are used to express the sets of users and resources to which a rule applies. 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 (in other words, a subset of $\text{Val}_s \setminus \{\perp\}$ or $\text{Val}_m \setminus \{\perp\}$, depending on whether a is single-valued or multi-valued). We refer to $e(a)$ as the *conjunct* for attribute a . A user-attribute expression e *uses* attribute a if $e(a) \neq \top$. Let $\text{attr}(e)$ denote the set of attributes used by e .

A user u *satisfies* a user-attribute expression e , denoted $u \models e$, iff $(\forall a \in A_{u,1}. e(a) = \top \vee \exists v \in e(a). d_u(u, a) = v)$ and $(\forall a \in A_{u,m}. e(a) = \top \vee \exists v \in e(a). d_u(u, a) \supseteq v)$. For multi-valued attributes, we use the condition $d_u(u, a) \supseteq v$ instead of $d_u(u, a) = v$ 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 $A_{u,1} = \{\text{dept}, \text{position}\}$ and $A_{u,m} = \{\text{courses}\}$. The function e_1 with $e_1(\text{dept}) = \{\text{CS}\}$ and $e_1(\text{position}) = \{\text{grad}, \text{ugrad}\}$ and $e_1(\text{courses}) = \{\{\text{CS101}, \text{CS102}\}\}$ 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.

We introduce a concrete syntax for attribute expressions, for improved readability in examples. We write a user attribute expression as a conjunction of the conjuncts not equal to \top . Suppose $e(a) \neq \top$. Let $v = e(a)$. When a is single-valued, we write the conjunct for a as $a \in v$; as syntactic sugar, if v is a singleton set $\{s\}$, we may write the conjunct as $a = s$. When a is multi-valued, we write the conjunct for a as $a \supseteq v$ (indicating that a is a superset of an element of v); as syntactic sugar, if v is a singleton set $\{s\}$, we may write the conjunct as $a \supseteq s$. For example, the above expression e_1 may be written as $\text{dept} = \text{CS} \wedge \text{position} \in \{\text{ugrad}, \text{grad}\} \wedge \text{courses} \supseteq \{\text{CS101}, \text{CS102}\}$. For an example that uses $\supseteq \in$, the expression e_2 that is the same as e_1 except with $e_2(\text{courses}) = \{\{\text{CS101}\}, \{\text{CS102}\}\}$ may be written as $\text{dept} = \text{CS} \wedge \text{position} \in \{\text{ugrad}, \text{grad}\} \wedge \text{courses} \supseteq \in \{\{\text{CS101}\}, \{\text{CS102}\}\}$, 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 $\llbracket e \rrbracket_U$, is the set of users in U that satisfy it: $\llbracket e \rrbracket_U = \{u \in U \mid u \models e\}$. User attribute data is an implicit argument to $\llbracket e \rrbracket_U$. We say that e *characterizes* the set $\llbracket e \rrbracket_U$.

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 $a_{u,m} \supseteq a_{r,m}$, $a_{u,m} \ni a_{r,1}$, or $a_{u,1} = a_{r,1}$, where $a_{u,1} \in A_{u,1}$, $a_{u,m} \in A_{u,m}$, $a_{r,1} \in A_{r,1}$, and $a_{r,m} \in A_{r,m}$. 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. Other forms of atomic constraint are possible (e.g., $a_{u,m} \subseteq a_{r,m}$) but less common, so we leave them for future work. Let $\text{uAttr}(f)$ and $\text{rAttr}(f)$ 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 $d_u(u, \text{uAttr}(f)) \neq \perp$ and $d_r(u, \text{rAttr}(f)) \neq \perp$ and formula f holds when the values $d_u(u, \text{uAttr}(f))$ and $d_r(u, \text{rAttr}(f))$ 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 $\langle e_u, e_r, O, c \rangle$, 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 $\rho = \langle e_u, e_r, O, c \rangle$, let $\text{uae}(\rho) = e_u$, $\text{rae}(\rho) = e_r$, $\text{ops}(\rho) = O$, and $\text{con}(\rho) = c$. For example, the rule $\langle \text{true}, \text{type}=\text{task} \wedge \text{proprietary}=\text{false}, \{\text{read}, \text{request}\}, \text{projects} \ni \text{project} \wedge \text{expertise} \supseteq \text{expertise} \rangle$ 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 $u \models \text{uae}(\rho) \wedge r \models \text{rae}(\rho) \wedge o \in \text{ops}(\rho) \wedge \langle u, r \rangle \models \text{con}(\rho)$.

An *ABAC policy* is a tuple $\langle U, R, Op, A_u, A_r, d_u, d_r, Rules \rangle$, 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 $\llbracket \rho \rrbracket = \{ \langle u, r, o \rangle \in U \times R \times Op \mid \langle u, r, o \rangle \models \rho \}$. Note that U, R, d_u , and d_r are implicit arguments to $\llbracket \rho \rrbracket$.

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

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 $I = \langle U, R, Op, A_u, A_r, d_u, d_r, L \rangle$, 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 $\pi = \langle U, R, Op, A_u, A_r, d_u, d_r, Rules \rangle$ 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) [20], 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 [2]. 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, $\text{WSC}(\langle e_u, e_r, O, c \rangle) = w_1 \text{WSC}(e_u) + w_2 \text{WSC}(e_r) + w_3 \text{WSC}(O) + w_4 \text{WSC}(c)$, 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 $\llbracket \pi \rrbracket$ 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 $\llbracket \pi \rrbracket \setminus UP(L)$. The under-assignments are $UP(L) \setminus \llbracket \pi \rrbracket$. 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 mined policy, and it improves the stability of the generated policy; consideration of noise is left for future work. We define a policy quality metric that is a weighted sum of these aspects:

$$Q_{\text{pol}}(\pi, L) = \text{WSC}(\pi) + w_o \left| \llbracket \pi \rrbracket \setminus UP(L) \right| / |U| \quad (1)$$

and for a set S of user-permission tuples, the frequency-weighted size of S with respect to log L is $|S|_L = \sum_{\langle u,r,o \rangle \in S} \text{freq}(\langle u,r,o \rangle, L)$, where the relative frequency of a user-permission tuple in a log is given by the *frequency function* $\text{freq}(\langle u,r,o \rangle, L) = |\{e \in L \mid \text{userPerm}(e) = \langle u,r,o \rangle\}| / |L|$, where the user-permission part of a log entry is given by $\text{userPerm}(\langle u,r,o,t \rangle) = \langle u,r,o \rangle$. Using the relative frequency, instead of the number of occurrences, of a user-permission tuple as the weight in the definition of frequency-weighted size allows w_u to be chosen independent of the duration of monitoring and the number of monitored users; for example, if the duration of monitoring or the number of monitored users is doubled, and the distribution of their behaviors and the value of w_u are kept unchanged, then the policy quality is unchanged. Similarly, dividing the number of over-assignments by $|U|$ allows w_o to be chosen independent of the number of users (and the duration of monitoring).

For simplicity, our presentation of the problem and algorithm assume that attribute data does not change during the time covered by the log. Accommodating 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 $\llbracket \rho \rrbracket_{\text{LE}}$, rather than $\llbracket \rho \rrbracket$. The definition of $\llbracket \rho \rrbracket_{\text{LE}}$ is similar to the definition of $\llbracket \rho \rrbracket$, except that, when determining whether a log entry is in $\llbracket \rho \rrbracket_{\text{LE}}$, 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 [39]. 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 $\text{addCandRule}(s_u, s_r, s_o, cc, \text{uncovUP}, \text{Rules})$ 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 $\text{generalizeRule}(\rho, cc, \text{uncovUP}, \text{Rules})$ to generalize rule $\rho = \langle e_u, e_r, s_o, \emptyset \rangle$ to ρ' and adds ρ' to candidate rule set Rules . The details of the functions called by addCandRule are described next.

The function $\text{computeUAE}(s, U)$ in Figure 3 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. After constructing a candidate expression e , it calls $\text{elimRedundantSets}(e)$, which attempts to lower the WSC of e by examining the conjunct for each multi-valued user attribute, and removing each set that is a superset of another set in the same conjunct; this leaves the meaning of the rule unchanged, because \supseteq is used in the condition for multi-valued attributes in the semantics of user attribute expressions. Pseudocode for elimRedundantSets is straightforward and omitted. The expression e_u returned by computeUAE might not be minimum-sized among expressions that characterize s : it is possible that some attributes mapped to a set of values by e_u can instead be mapped to \top . We defer minimization of e_u until after the call to generalizeRule (described below), because minimizing e_u before that would reduce opportunities to find relations between values of user attributes and resource attributes in generalizeRule .

The function $\text{computeRAE}(s, R)$ computes a resource-attribute expression that characterizes the set s of resources. The definition is the same as for computeUAE , except using resource attributes instead of user

attributes, and the call to `elimRedundantSets` is omitted. Pseudocode for `computeRAE` is 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 4 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*.

$$Q_{\text{rul}}(\rho, UP) = \frac{|\llbracket \rho \rrbracket \cap UP|}{|\rho|} \times \left(1 - \frac{w'_o \times |\llbracket \rho \rrbracket \setminus UP(L)|}{|\llbracket \rho \rrbracket|}\right).$$

In `generalizeRule`, $uncovUP$ is the second argument to Q_{rul} , so $\llbracket \rho \rrbracket \cap UP$ 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 $Q_{\text{rul}}^{\text{freq}}$ 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 [23]. Specifically, the definition of $Q_{\text{rul}}^{\text{freq}}$ is obtained from the definition of Q_{rul} by replacing $|\llbracket \rho \rrbracket \cap UP|$ with $|\llbracket \rho \rrbracket \cap UP|_L$ (recall that $|\cdot|_L$ is defined in Section 3).

We also developed a rule quality metric $Q_{\text{rul}}^{\text{ILP}}$ based closely on the theory quality metric for inductive logic programming described in [26]. Details of the definition appear in Appendix A.

The function `mergeRules($Rules$)` in Figure 4 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 $\llbracket \rho \rrbracket$ is also in $\llbracket \rho' \rrbracket$. 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 ρ_{mrg} does not have over-assignments, 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.

```

// Rules is the set of candidate rules
Rules =  $\emptyset$ 
// uncovUP contains user-permission tuples
// in  $UP_0$  that are not covered by Rules
uncovUP =  $UP_0$ .copy()
while  $\neg$ uncovUP.isEmpty()
  // Select an uncovered tuple as a "seed".
   $\langle u, r, o \rangle$  = some tuple in uncovUP
  cc = candidateConstraint(r, u)
  //  $s_u$  contains users with permission  $\langle r, o \rangle$ 
  // and that have the same candidate
  // constraint for r as u
   $s_u = \{u' \in U \mid \langle u', r, o \rangle \in UP_0$ 
     $\wedge$  candidateConstraint(r, u') = cc}
  addCandRule( $s_u$ , {r}, {o}, cc, uncovUP, Rules)
  //  $s_o$  is set of operations that u can apply to r
   $s_o = \{o' \in Op \mid \langle u, r, o' \rangle \in UP_0\}$ 
  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' =  $\emptyset$ 
Repeatedly move highest-quality rule
from Rules to Rules' until
 $\sum_{\rho \in Rules'} \llbracket \rho \rrbracket \supseteq UP_0$ , using
 $UP_0 \setminus \llbracket Rules' \rrbracket$  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'

```

Figure 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  $\rho$  that covers user-permission tuples  $\{\langle u, r, o \rangle \mid u \in s_u \wedge r \in s_r \wedge o \in s_o\}$ .
 $e_u$  = computeUAE( $s_u, U$ );
 $e_r$  = computeRAE( $s_r, R$ );
 $\rho = \langle e_u, e_r, s_o, \emptyset \rangle$ 
 $\rho' =$  generalizeRule( $\rho, cc, uncovUP, Rules$ );
Rules.add( $\rho'$ );
uncovUP.removeAll( $\llbracket \rho' \rrbracket$ )

```

Figure 2: Compute a candidate rule ρ' and add ρ' to candidate rule set $Rules$

5 Functions to Simplify Rules

The function `simplifyRules($Rules$)` in Figure 5 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 ways, which are embodied in the following functions that it calls. The names of these functions start with “elim”, because they attempt to eliminate unnecessary parts of rules. To enable `simplifyRules` to determine whether any rules were simplified, each “elim” function returns a Boolean value indicating whether it simplified any rules. For brevity, computation of the Boolean return values of “elim” functions are not reflected in the pseudocode.

The function `elimRedundantSets` is described above. It returns false, even if some redundant sets were eliminated, because elimination of redundant sets does not affect the meaning or mergeability of rules, so it need not trigger another iteration of merging and simplification.

The function `elimConjuncts($\rho, Rules, UP$)` in Figure 5 attempts to increase the quality of rule ρ by eliminating some of the conjuncts. It calls the function `elimConjunctsHelper(ρ, A, UP)` in Figure 5, which considers all rules that differ from ρ by mapping a subset A' of the tagged attributes in A to \top instead of to a set of values; among the resulting rules that are valid, it returns one with the highest quality. A

```

function computeUAE( $s, U$ )
// First try to characterize set  $s$  of users without using
// uid. Use all other attributes which have known
// values for all users in  $s$ .
 $e = (\lambda a \in A_u.$ 
     $a = \text{uid} \vee (\exists u \in s. d_u(u, a) = \perp) ? \top : \bigcup_{u \in s} d_u(u, a))$ 
if  $[e]_U \neq s$ 
    // uid is needed to characterize  $s$ 
     $e(\text{uid}) = \bigcup_{u \in s} d_u(u, \text{uid})$ 
end if
elimRedundantSets( $e$ )
return  $e$ 

```

Figure 3: Compute a user-attribute expression that characterizes set s of users, where U is the set of all users.

tagged attribute is a pair of the form $\langle \text{"user"}, a \rangle$ with $a \in A_u$ or $\langle \text{"res"}, a \rangle$ with $a \in A_r$. The set A_{unrm} in `elimConjuncts` is a set of *unremovable* tagged attributes; it is a parameter of the algorithm, specifying attributes that should not be eliminated, because eliminating them increases the risk of generating an overly general policy, i.e., a policy that might grant inappropriate permissions when new users or new resources (hence new permissions) are added to the system. We use a combinatorial algorithm for `elimConjuncts` that evaluates all combinations of conjuncts that can be eliminated, because elimination of one conjunct might prevent elimination of another conjunct. This algorithm makes `elimConjuncts` worst-case exponential in the numbers of user attributes and resource attributes that can be eliminated while preserving validity of the rule; in practice the number of such attributes is small. `elimConjuncts` also considers whether to remove conjuncts from the user attribute expression or the resource attribute expression first, because elimination of a conjunct in one attribute expression might prevent elimination of a conjunct in the other. The algorithm could simply try both orders, but instead it uses a heuristic that, in our experiments, is faster and almost as effective: if $\text{maxConjunctSz}(e_u) \geq \text{maxConjunctSz}(e_r)$ then eliminate conjuncts from the user attribute expression first, otherwise eliminate conjuncts from the resource attribute expression first, where $\text{maxConjunctSz}(e)$ is the size (WSC) of the largest conjunct in attribute expression e . `elimConjunctsHelper` calls the function `elimAttribute(ρ, ta)`, which returns a copy of rule ρ with the conjunct for tagged attribute ta removed from the user attribute expression or resource attribute expression as appropriate (in other words, the specified attribute is mapped to \perp); pseudocode for `elimAttribute` is straightforward and omitted.

The function `elimConstraints($\rho, Rules, UP$)` attempts to improve the quality of ρ by removing unnecessary atomic constraints from ρ 's constraint. An atomic constraint is *unnecessary* in a rule ρ if removing it from ρ 's constraint leaves ρ valid. Pseudocode for `elimConstraints` is analogous to `elimConjuncts`, except it considers removing atomic constraints instead of conjuncts from rules.

The function `elimElements(ρ)` attempts to decrease the WSC of rule ρ by removing elements from sets in conjuncts for multi-valued user attributes, if removal of those elements preserves validity of ρ ; note that, because \subseteq is used in the semantics of user attribute expressions, the set of user-permission tuples that satisfy a rule is never decreased by such removals. It would be reasonable to use a combinatorial algorithm for `elimElements`, in the same style as `elimConjuncts` and `elimConstraints`, because elimination of one set element can prevent elimination of another. We decided to use a simple linear algorithm for this function, for simplicity and because it is likely to give the same results, because `elimElements` usually eliminates only 0 or 1 set elements per rule in our experiments. Pseudocode for `elimElements` is straightforward and omitted.

The function `elimOverlapVal($\rho, Rules$)` attempts to decrease the WSC of rule ρ by removing values from conjuncts of attribute expressions in ρ if there are other rules that cover the affected user-permission tuples


```

function generalizeRule( $\rho$ ,  $cc$ ,  $uncovUP$ ,
                         $Rules$ )
//  $\rho_{best}$  is best generalization of  $\rho$ 
 $\rho_{best} = \rho$ 
//  $gen[i][j]$  is a generalization of  $\rho$  using
//  $cc[i]$ 
 $gen = \text{new Rule}[cc.length][3]$ 
for  $i = 1$  to  $cc.length$ 
   $f = cc[i]$ 
  // generalize by adding  $f$  and eliminating
  // conjuncts for both attributes used in  $f$ .
   $gen[i][1] = \langle \text{uae}(\rho)[\text{uAttr}(f) \mapsto \top],$ 
                 $\text{rae}(\rho)[\text{rAttr}(f) \mapsto \top],$ 
                 $\text{ops}(\rho), \text{con}(\rho) \cup \{f\} \rangle$ 
  // generalize by adding  $f$  and eliminating
  // conjunct for user attribute used in  $f$ 
   $gen[i][2] = \langle \text{uae}(\rho)[\text{uAttr}(f) \mapsto \top], \text{rae}(\rho),$ 
                 $\text{ops}(\rho), \text{con}(\rho) \cup \{f\} \rangle$ 
  // generalize by adding  $f$  and eliminating
  // conjunct for resource attrib. used in  $f$ .
   $gen[i][3] = \langle \text{uae}(\rho), \text{rae}(\rho)[\text{rAttr}(f) \mapsto \top],$ 
                 $\text{ops}(\rho), \text{con}(\rho) \cup \{f\} \rangle$ 
end for
for  $i = 1$  to  $cc.length$  and  $j = 1$  to  $3$ 
  // try to further generalize  $gen[i]$ 
   $\rho'' = \text{generalizeRule}(gen[i][j], cc[i+1 ..],$ 
                           $uncovUP, Rules)$ 
  if  $Q_{rul}(\rho'', uncovUP) > Q_{rul}(\rho_{best},$ 
                                      $uncovUP)$ 
     $\rho_{best} = \rho''$ 
  end if
end for
return  $\rho_{best}$ 

function mergeRules( $Rules$ )
// Remove redundant rules
 $redun = \{\rho \in Rules \mid \exists \rho' \in Rules \setminus \{\rho\}.$ 
           $\llbracket \rho \rrbracket \cap UP_0 \subseteq \llbracket \rho' \rrbracket \cap UP_0\}$ 
 $Rules.removeAll(redun)$ 
// Merge rules
 $workSet = \{(\rho_1, \rho_2) \mid \rho_1 \in Rules \wedge \rho_2 \in Rules$ 
             $\wedge \rho_1 \neq \rho_2 \wedge \text{con}(\rho_1) = \text{con}(\rho_2)\}$ 
while  $\text{not}(workSet.empty())$ 
   $(\rho_1, \rho_2) = workSet.remove()$ 
   $\rho_{mrg} = \langle \text{uae}(\rho_1) \cup \text{uae}(\rho_2),$ 
                $\text{rae}(\rho_1) \cup \text{rae}(\rho_2),$ 
                $\text{ops}(\rho_1) \cup \text{ops}(\rho_2), \text{con}(\rho_1) \rangle$ 
  // Find rules that become redundant
  // if merged rule  $\rho_{mrg}$  is added
   $redun = \{\rho \in Rules \mid \llbracket \rho \rrbracket \subseteq \llbracket \rho_{mrg} \rrbracket\}$ 
  // Add the merged rule and remove redun-
  // dant rules if this improves policy quality
  // and the merged rule does not have
  // over-assignments.
  if  $(Q_{pol}(Rules \cup \{\rho_{mrg}\} \setminus redun) < Q_{pol}(Rules)$ 
       $\wedge \llbracket \rho_{mrg} \rrbracket \subseteq UP_0)$ 
     $Rules.removeAll(redun)$ 
     $workSet.removeAll(\{(\rho_1, \rho_2) \in workSet \mid$ 
                        $\rho_1 \in redun \vee \rho_2 \in redun\})$ 
     $workSet.addAll(\{(\rho_{mrg}, \rho) \mid \rho \in Rules$ 
                     $\wedge \text{con}(\rho) = \text{con}(\rho_{mrg})\})$ 
     $Rules.add(\rho_{mrg})$ 
  end if
end while
return true if any rules were merged

```

Figure 4: 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 $e(a) = e_1(a) \cup e_2(a)$.

and have higher quality. Specifically, a value v in the conjunct for a user attribute a in ρ is removed if there is another rule ρ' in $Rules$ such that (1) $\text{attr}(\text{uae}(\rho')) \subseteq \text{attr}(\text{uae}(\rho))$ and $\text{attr}(\text{rae}(\rho')) \subseteq \text{attr}(\text{rae}(\rho))$, (2) the conjunct of $\text{uae}(\rho')$ for a contains v , (3) each conjunct of $\text{uae}(\rho')$ or $\text{rae}(\rho')$ other than the conjunct for a is either \top or a superset of the corresponding conjunct of ρ , and (4) $\text{con}(\rho') \subseteq \text{con}(\rho)$. The condition for removal of a value in the conjunct for a resource attribute is analogous. If a conjunct of $\text{uae}(\rho)$ or $\text{rae}(\rho)$ becomes empty, ρ is removed from $Rules$. $\text{elimOverlapVal}(\rho, Rules)$ returns true if it modifies or removes ρ , otherwise it returns false. Pseudocode for elimOverlapVal is straightforward and omitted.

The function $\text{elimOverlapOp}(\rho, Rules)$ attempts to decrease the WSC of rule ρ by removing operations from $\text{ops}(\rho)$, if there are other rules that cover the affected user-permission tuples. Specifically, an operation

o is removed from $\text{ops}(\rho)$ if there is another rule ρ' in $Rules$ such that (1) $\text{attr}(\text{uae}(\rho')) \subseteq \text{attr}(\text{uae}(\rho))$ and $\text{attr}(\text{rae}(\rho')) \subseteq \text{attr}(\text{rae}(\rho))$, (2) $\text{ops}(\rho')$ contains o , (3) each conjunct of $\text{uae}(\rho')$ or $\text{rae}(\rho')$ is either \top or a superset of the corresponding conjunct of ρ , and (4) $\text{con}(\rho')$ is a subset of $\text{con}(\rho)$. If $\text{ops}(\rho)$ becomes empty, ρ is removed from $Rules$. $\text{elimOverlapOp}(\rho, Rules)$ returns true if it modifies or removes ρ , otherwise it returns false. Pseudocode for elimOverlapOp is straightforward and omitted.

5.1 Example

We illustrate the algorithm on a small fragment of our university case study (*cf.* Section 6.1). The fragment contains a single rule $\rho_0 = \langle \text{true}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{crsTaught} \ni \text{crs} \rangle$ 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: $\{\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore}, t_1 \rangle, \langle \text{csFac2}, \text{cs601gradebook}, \text{readScore}, t_2 \rangle, \langle \text{csStu3}, \text{cs601gradebook}, \text{addScore}, t_3 \rangle\}$. 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 = 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 $\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore} \rangle$ as the first seed, and calls function $\text{candidateConstraint}$ 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 $e_u = (\text{position} \in \{\text{faculty}, \text{student}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{crsTaught} \supseteq \{\{\text{cs601}\}\})$. addCandRule also calls computeRAE to compute a resource-attribute expression that characterizes {cs601gradebook}; the result is $e_r = (\text{crs} \in \{\text{cs601}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{type} \in \{\text{gradebook}\})$. The set of operations considered in this call to addCandRule is simply $s_o = \{\text{addScore}\}$. 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 entitlement in 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 entitlement in 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 $\rho_3 = \langle \text{position} \in \{\text{faculty}, \text{student}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$. ρ_3 is discarded because it introduces an over-assignment user-permission tuple $\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore} \rangle$ 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.

6 Evaluation Methodology

We evaluate our policy mining algorithms on synthetic operation logs generated from ABAC policies (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.

```

function simplifyRules(Rules)
for  $\rho$  in Rules
  elimRedundantSets(uae( $\rho$ ))
  elimConjuncts( $\rho$ , Rules,  $UP_0$ )
  elimElements( $\rho$ )
end for
for  $\rho$  in Rules
  elimOverlapVal( $\rho$ , Rules)
end for
for  $\rho$  in Rules
  elimOverlapOp( $\rho$ , Rules)
end for
for  $\rho$  in Rules
  elimConstraints( $\rho$ , Rules,  $UP_0$ )
end for
return true if any  $\rho$  in Rules was changed

function elimConjuncts( $\rho$ , Rules, UP)
Au = {"user"}  $\times$  attr(uae( $\rho$ ))  $\setminus$   $A_{\text{unrm}}$ 
Ar = {"res"}  $\times$  attr(rae( $\rho$ ))  $\setminus$   $A_{\text{unrm}}$ 
if maxConjunctSz(uae( $\rho$ ))  $\geq$  maxConjunctSz(rae( $\rho$ ))
   $\rho'$  = elimConjunctsHelper( $\rho$ , Au, UP)
   $\rho''$  = elimConjunctsHelper( $\rho'$ , Ar, UP)
else
   $\rho'$  = elimConjunctsHelper( $\rho$ , Ar, UP)
   $\rho''$  = elimConjunctsHelper( $\rho'$ , Au, UP)
end if
if  $\rho'' \neq \rho$ 
  replace  $\rho$  with  $\rho''$  in Rules
end if

function elimConjunctsHelper( $\rho$ , A, UP)
 $\rho_{\text{best}} = \rho$ 
// Discard tagged attributes ta such that elimi-
// nation of the conjunct for ta makes  $\rho$  invalid.
for ta in A
   $\rho' = \text{elimAttribute}(\rho, ta)$ 
  if not  $\llbracket \rho' \rrbracket \subseteq UP_0$ 
    A.remove(ta)
  end if
end for
for i = 1 to A.length // treat A as an array
   $\rho' = \text{elimAttribute}(\rho, A[i])$ 
   $\rho'' = \text{elimConjunctsHelper}(\rho', A[i+1 ..])$ 
  if  $Q_{\text{rul}}(\rho'', UP) > Q_{\text{rul}}(\rho_{\text{best}}, UP)$ 
     $\rho_{\text{best}} = \rho''$ 
  end if
end for
return  $\rho_{\text{best}}$ 

```

Figure 5: Functions used to simplify rules.

6.1 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 [39].

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 [39]. 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).

6.2 Log Generation

We first describe a straightforward but inefficient log generation algorithm, and then describe the log summary generation algorithm that is actually used in our experiments.

Log Generation Algorithm. 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. We use completeness as a measure of log size relative to policy size. The algorithm generates each log entry by first selecting an ABAC rule, according to a probability distribution on roles, and then selecting a user-permission tuple that satisfies the rule, according to probability distributions on users, resources, and operations. This process is repeated until the specified completeness is reached.

Specifically, our log generation algorithm uses the following probability distributions. For each rule ρ in π , $P_{\text{rule}}(\rho)$ is the probability of selecting ρ . For each operation o and rule ρ , $P_{\text{op}}(o|\rho)$ is the conditional probability of selecting operation o when ρ has been selected; for sanity, we require $P_{\text{op}}(o|\rho) = 0$ if $o \notin \text{ops}(\rho)$. For each user u , resource r , and rule ρ , $P_{\text{ur}}(u, r|\rho)$ is the conditional probability of selecting user

u and resource r when rule ρ and resource r have been selected; for sanity, we require $P_{\text{ur}}(u, r|\rho) = 0$ if $u \not\models \text{uae}(\rho) \vee r \not\models \text{rae}(\rho) \vee \langle u, r \rangle \not\models \text{con}(\rho)$. The algorithm generates each log entry as follows: select a rule ρ based on $P_{\text{rule}}(\rho)$, select an operation o based on $P_{\text{op}}(o|\rho)$ and a user-permission tuple $\langle u, r \rangle$ based on $P_{\text{ur}}(u, r|\rho)$, and then create a log entry $\langle u, r, o, i \rangle$, where the unique identifier i is the index of this log entry in the synthetic log.

For convenience, we allow the user to specify probability distributions $P_{\text{user}}(u)$, $P_{\text{res}}(r)$, and $P'_{\text{op}}(o)$ instead of P_{ur} and P_{op} , and we compute P_{ur} and P_{op} from them as described below; intuitively, the probability associated with invalid values is distributed uniformly among the valid values by this computation. These alternative probability distributions provide less detailed control but are simpler and more convenient.

$$\begin{aligned}
 P_{\text{ur}}(u, r|\rho) &= \begin{cases} c^{-1}P_{\text{user}}(u)P_{\text{res}}(r) & \text{if } \langle u, r \rangle \in \llbracket \rho \rrbracket_{\text{ur}} \\ 0 & \text{otherwise} \end{cases} \\
 &\text{where } c = \sum_{\langle u, r \rangle \in \llbracket \rho \rrbracket_{\text{ur}}} P_{\text{user}}(u)P_{\text{res}}(r) \\
 P_{\text{op}}(o|\rho) &= \begin{cases} c^{-1}P'_{\text{op}}(o) & \text{if } o \in \text{ops}(\rho) \\ 0 & \text{otherwise} \end{cases} \\
 &\text{where } c = \sum_{o \in \text{ops}(\rho)} P'_{\text{op}}(o)
 \end{aligned}$$

where the user-resource part of the meaning of a rule is $\llbracket \rho \rrbracket_{\text{ur}} = \bigcup_{\langle u, r, o \rangle \in \llbracket \rho \rrbracket} \{\langle u, r \rangle\}$.

Log Summary Generation Algorithm. When the probability distributions give much higher probability to some user-permission tuples than others, using the above algorithm to generate logs with high completeness is very inefficient, because very long logs, with many entries for high-probability user-permission tuples, are generated. 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. Specifically, instead of generating logs, we directly generate *log summaries*, which are simply sets of user-permission tuples with associated frequencies that sum to 1 (in other words, a log summary defines a set of user-permission tuples and a frequency function). For a given ABAC policy and probability distributions, we generate a log summary with completeness 1 using a program that computes the asymptotic frequency with which each tuple in UP_0 would appear in an infinitely long log generated using the above algorithm. To generate a series of log summaries with varying completeness, we start with the log summary with completeness 1 and repeatedly apply the following procedure that generates a log summary of lower completeness from a log summary with higher completeness: select the appropriate number of user-permission tuples from the latter, with the selection probability for each tuple equal to its associated frequency, and then normalize the frequencies of the selected tuples so they sum to 1.

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).

6.3 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 6.2 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 $S_{\text{syn}}^u(e, e') = |A_u|^{-1} \sum_{a \in A_u} J(e(a), e'(a))$. Syntactic similarity of RAEs is defined by $S_{\text{syn}}^r(e, e') = |A_r|^{-1} \sum_{a \in A_r} J(e(a), e'(a))$. The syntactic similarity of rules $\langle e_u, e_r, O, c \rangle$ and $\langle e'_u, e'_r, O', c' \rangle$ is the average of the similarities of their components, specifically, the average of $S_{\text{syn}}^u(e_u, e'_u)$, $S_{\text{syn}}^r(e_r, e'_r)$, $J(O, O')$, 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). Formally,

$$S_{\text{syn}}(Rules, Rules') = |Rules|^{-1} \sum_{\rho \in Rules} \min(\{S_{\text{syn}}(\rho, \rho') \mid \rho' \in Rules'\})$$

$$S_{\text{syn}}(\pi, \pi') = \max(S_{\text{syn}}(\text{rules}(\pi), \text{rules}(\pi')), S_{\text{syn}}(\text{rules}(\pi'), \text{rules}(\pi)))$$

Syntactic similarity ranges from 0 (completely different) to 1 (identical). High syntactic similarity between the original and generated ABAC policies is not always desirable, e.g., for synthetic policies, which tend to have unnecessarily complicated rules.

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.

7 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 6 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 (i.e., syntactic similarity and semantic similarity are 1), and the semantics of synthetic policies is reconstructed almost exactly: the semantic similarity is 0.99. This is a non-trivial result, especially for the case studies: an algorithm could easily generate a policy with over-assignments or 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 [23]. For the experiments in [23, 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 0.91 for log completeness 60% or higher, and is above 0.94 for log completeness 70% 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 0.99 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 0.85 for log completeness 60% or higher, and above 0.94 for log completeness 80% or higher. These results are quite good, in the sense that our algorithm compensates for most of the log incompleteness. For example, at log completeness 0.6, for policies generated by a policy mining algorithm that produces policies granting exactly the entitlements reflected in the log, the semantic similarity would be 0.6. With our algorithm, the semantic similarity, averaged over the 5 examples, is 0.95. Thus, in this case, our algorithm compensates for $35/40 = 87.5\%$ of the incompleteness.

The fractions of over-assignments are below 0.03 for log completeness 60% or higher. The fractions of under-assignments are below 0.05 for log completeness 60% or higher for the case studies and are below 0.05 for log completeness 80% or higher for synthetic policies. The graphs also show that the semantic differences are due more to under-assignments than over-assignments; this is desirable from a security perspective.

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. Q_{rul} is moderately better overall than $Q_{\text{rul}}^{\text{freq}}$ and significantly better overall than $Q_{\text{rul}}^{\text{ILP}}$.

Comparison with Inductive Logic Programming. To translate ABAC policy mining from logs to Progol [24, 27], we used the translation of ABAC policy mining from ACLs to Progol in [39, Sections 5.5, 16], except negative examples corresponding to absent user-permission tuples are omitted from the generated program, and the statement `set(possibly)?` is included, telling Progol to use its algorithm for learning from

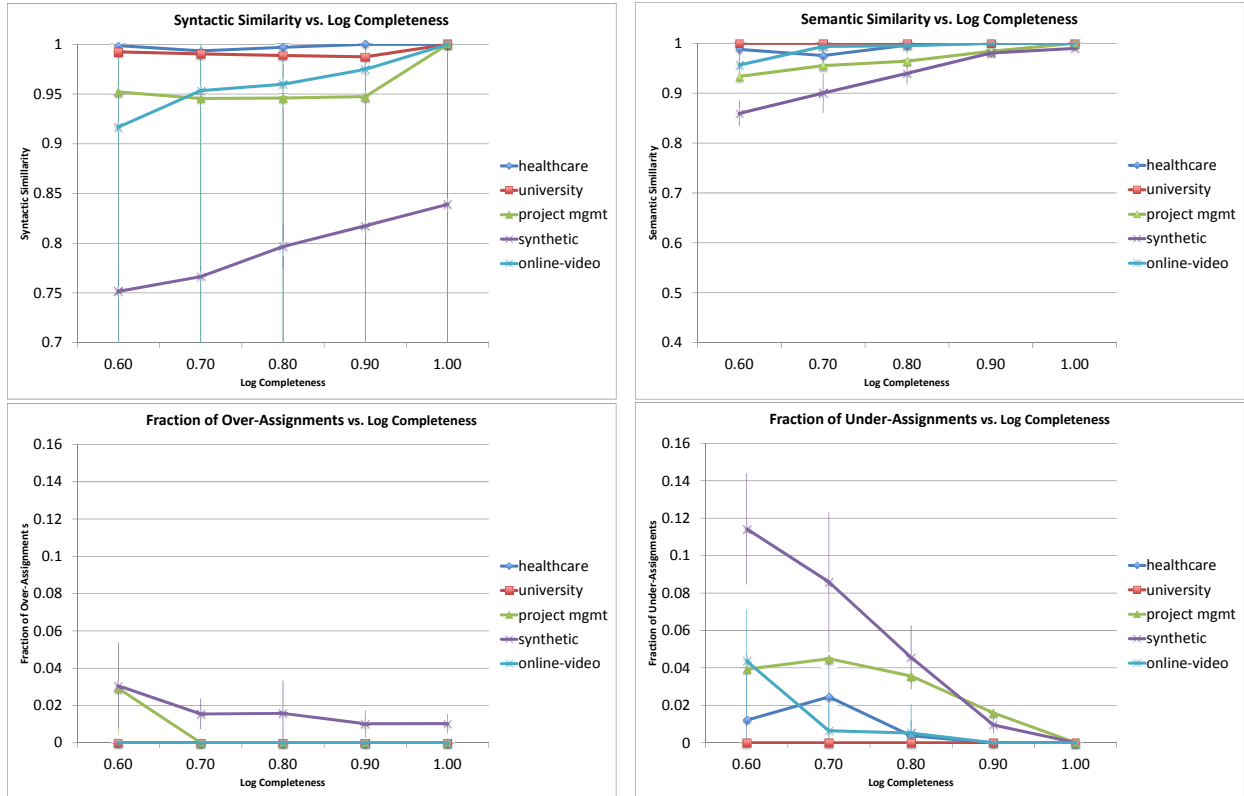


Figure 6: Top: Syntactic similarity and semantic similarity of original and mined ABAC policies, as a function of log completeness. Bottom: Fractions of over-assignments and under-assignments in mined ABAC policy, as a function of log completeness.

positive examples. For the four case studies with manually written attribute data (in contrast, Figure 6 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. We used Progol 4.4 for these experiments, because Progol 5.0 segmentation-faults whenever the `posonly` option is set. We also ran the experiments without the `posonly` option using Progol 4.4 and Progol 5.0, and we obtained the same results.

Comparison with Author-Topic Model. Inspired by Molloy *et al.*'s translation of RBAC policy mining from logs to the problem of constructing an Author-Topic Model (ATM) [23], we developed and implemented a translation of ABAC policy mining from logs to ATM. In our translation, (1) topics correspond to rules, (2) authors correspond to tuples containing a user attribute expression, a resource attribute expression, and a constraint that can appear together in a candidate ABAC rule, (3) documents correspond to a user-resource pair, and (4) words correspond to operations. We used Steyvers and Griffiths' implementation of the ATM algorithm, available at http://psiexp.ss.uci.edu/research/programs_data/toolbox.htm. We did not implement the discretization algorithm based on simulated annealing to compute an ABAC policy from the generated author-topic model; we manually considered the top few authors for each document. We applied this algorithm to the four case studies with manually written attribute data, using generated logs with completeness ranging from 0.6 to 1 in steps of 0.1, and using 10 logs for each completeness level. In each case, we specified the number of rules in the original policy as the number of topics. In all cases (i.e., for all

logs at each log completeness level for each case study), the semantic similarity of the original and mined policies is below 0.6. The reason for the low semantic similarity is that the authors selected by the ATM algorithm correspond to narrow rules, not general rules applicable to multiple projects, departments, wards, or whatever.

8 Related Work

We are not aware of prior work on ABAC mining from logs. We discuss prior work on related problems.

Our policy mining algorithm is based on our algorithm for ABAC policy mining from ACLs [39]. The main differences are described in Section 1. The fact that our algorithm can be adapted to work well for this new problem, without changing the algorithm’s overall structure, demonstrates the power and flexibility of our approach.

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 [1] and are supported by statistically strong evidence. They are not designed to produce a set of rules 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 (to reduce overall policy size).

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [28]. 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.

Gal-Oz *et al.* [10] mine roles from logs that record sets of permissions exercised together in one high-level operation. Their 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. They evaluate their algorithm on synthetic logs. 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 RBAC policy that is most likely to generate the behavior (usage of permissions) observed in the logs [23]. They give an algorithm, based on Rosen-Zvi *et al.*’s algorithm for learning Author-Topic Models (ATMs) [29], to mine meaningful roles from logs and attribute data, i.e., roles such that the user-role assignment is statistically correlated with user attributes. A discretization algorithm based on simulated annealing is used to produce an RBAC policy from the probabilistic results of the ATM algorithm. We adapted their approach to ABAC policy mining from logs as described in Section 7, and found that it is less effective than our algorithm on the small examples we tried. Another issue with this approach is scalability: the adapted algorithm enumerates all authors, and the number of authors is very large, because authors correspond to tuples containing a well-formed UAE, RAE and constraint (i.e., tuples with the components of a candidate rule other than the operation set). In contrast, our algorithm never enumerates all well-formed candidate rules; it constructs candidate rules by generalizing specific rules constructed from seeds in the input. The number of authors is much smaller in Molloy *et al.*’s work, which does not consider resource attributes, and hence an author corresponds to only a candidate user attribute expression. Another issue with this approach is that it requires specifying the desired number of rules, which is difficult to predict.

Zhang *et al.* apply the experience-based access management approach [11] for role evolution [43]. In particular, they use machine learning algorithms based on Näive Bayes classifiers to assess the quality of the roles in a given role hierarchy relative to given access logs. The quality is based on the strength of the correlation between the user-role assignment and users' access patterns. The analysis results can be used to improve the role hierarchy. Zhang *et al.* also propose an algorithm based on Support Vector Machines to evolve a role hierarchy [42]. Given a role hierarchy and access logs, the algorithm tries to generate a new role hierarchy that optimizes a policy quality metric based on both the distance between the given policy and the generated policy and the homogeneity of the generated roles with respect to the logs. Role homogeneity is a measure of the similarity of the access patterns of the members of a role. These papers do not consider improvement or mining of ABAC policies.

9 Conclusion

This paper presents an algorithm for mining ABAC policies from logs and attribute data. Experiments with case studies and synthetic policies demonstrate the 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.

Directions for future work include handling additional policy language constructs such as arithmetic inequalities and negation, better automated tuning of parameters such as w'_o , characterization of the algorithm's effectiveness as a function of input characteristics such as the ratio of frequencies of the most and least frequent items (*cf.* Section 6.2) and the complexity of the ABAC policy (average number of conjuncts per rule, etc.), and experiments with noisy logs.

Acknowledgments. Christian Hesselbach did the comparison with Progol.

References

- [1] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. 20th International Conference on Very Large Data Bases (VLDB). pp. 487–499 (1994)
- [2] Beckerle, M., Martucci, L.A.: Formal definitions for usable access control rule sets—From goals to metrics. In: Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS). pp. 2:1–2:11. ACM (2013)
- [3] Colantonio, A., Di Pietro, R., Ocello, A.: A cost-driven approach to role engineering. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC). pp. 2129–2136. ACM (2008)
- [4] Colantonio, A., Di Pietro, R., Ocello, A., Verde, N.V.: A formal framework to elicit roles with business meaning in RBAC systems. In: Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 85–94 (2009)
- [5] Colantonio, A., Di Pietro, R., Verde, N.V.: A business-driven decomposition methodology for role mining. *Computers & Security* 31(7), 844–855 (October 2012)
- [6] Federal Chief Information Officer Council: Federal identity credential and access management (ficam) roadmap and implementation guidance, version 2.0 (Dec 2011), <http://www.idmanagement.gov/documents/ficam-roadmap-and-implementation-guidance>

- [7] Frank, M., Basin, D.A., Buhmann, J.M.: A class of probabilistic models for role engineering. In: Proceedings of the 2008 ACM Conference on Computer and Communications Security. pp. 299–310. ACM (2008)
- [8] Frank, M., Buhmann, J.M., Basin, D.A.: On the definition of role mining. In: Proceedings of the 15th ACM Symposium on Access Control Models and Technologies. pp. 35–44. ACM (2010)
- [9] Frank, M., Buhmann, J.M., Basin, D.A.: Role mining with probabilistic models. *ACM Transactions on Information and System Security* 15(4), 1–28 (2013)
- [10] 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)
- [11] Gunter, C.A., Liebovitz, D.M., Malin, B.: Experience-based access management: A life-cycle framework for identity and access management systems. *IEEE Security & Privacy* 9(5), 48–55 (2011)
- [12] Guo, Q., Vaidya, J., Atluri, V.: The role hierarchy mining problem: Discovery of optimal role hierarchies. In: Proc. 2008 Annual Computer Security Applications Conference (ACSAC). pp. 237–246. IEEE Computer Society (2008)
- [13] Hachana, S., Cuppens-Bouahia, 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)
- [14] 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), http://csrc.nist.gov/publications/drafts/800-162/sp800_162_draft.pdf
- [15] Kuhlmann, M., Shohat, D., Schimpf, G.: Role mining—revealing business roles for security administration using data mining technology. In: Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 179–186. ACM (2003)
- [16] Lu, H., Vaidya, J., Atluri, V.: Optimal Boolean matrix decomposition: Application to role engineering. In: Proc. 24th International Conference on Data Engineering (ICDE). pp. 297–306. IEEE (2008)
- [17] Lu, H., Vaidya, J., Atluri, V., Hong, Y.: Constraint-aware role mining via extended Boolean matrix decomposition. *IEEE Trans. Dependable Sec. Comput.* 9(5), 655–669 (2012)
- [18] Ma, X., Li, R., Lu, Z.: Role mining based on weights. In: Proceedings of the 15th ACM Symposium on Access Control Models and Technologies. pp. 65–74 (2010)
- [19] Mitra, B., Sural, S., Atluri, V., Vaidya, J.: Toward mining of temporal roles. In: Proc. 27th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec). Lecture Notes in Computer Science, vol. 7964, pp. 65–80. Springer (2013)
- [20] 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)
- [21] Molloy, I., Li, N., Li, T., Mao, Z., Wang, Q., Lobo, J.: Evaluating role mining algorithms. In: Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 95–104. ACM (2009)

- [22] Molloy, I., Li, N., Qi, Y.A., Lobo, J., Dickens, L.: Mining roles with noisy data. In: Proceeding of the 15th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 45–54. ACM (2010)
- [23] 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). pp. 45–56. ACM (2012)
- [24] Muggleton, S.: Learning from positive data. In: Proceedings of the 6th International Workshop on Inductive Logic Programming (ILP '96). Lecture Notes in Computer Science, vol. 1314, pp. 358–376. Springer (1997)
- [25] Muggleton, S., Bryant, C.H.: Theory completion using inverse entailment. In: Cussens, J., Frisch, A.M. (eds.) Proc. 10th International Conference on Inductive Logic Programming (ILP). pp. 130–146. Springer (2000)
- [26] Muggleton, S.H.: Inverse entailment and prolog. *New Generation Computing* 13, 245–286 (1995)
- [27] Muggleton, S.H., Firth, J.: CProlog4.4: a tutorial introduction. In: Dzeroski, S., Lavrac, N. (eds.) *Relational Data Mining*, pp. 160–188. Springer-Verlag (2001)
- [28] 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)
- [29] Rosen-Zvi, M., Chemudugunta, C., Griffiths, T., Smyth, P., Steyvers, M.: Learning author-topic models from text corpora. *ACM Transactions on Information Systems* 28(1), 4:1–4:38 (Jan 2010)
- [30] Schlegelmilch, J., Steffens, U.: Role mining with ORCA. In: Proceedings of the 10th ACM Symposium on Access Control Models and Technologies. pp. 168–176. ACM (2005)
- [31] Takabi, H., Joshi, J.B.D.: StateMiner: an efficient similarity-based approach for optimal mining of role hierarchy. In: Proc. 15th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 55–64. ACM (2010)
- [32] Vaidya, J., Atluri, V., Guo, Q.: The role mining problem: finding a minimal descriptive set of roles. In: Proc. 12th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 175–184. ACM (2007)
- [33] Vaidya, J., Atluri, V., Guo, Q., Adam, N.: Migrating to optimal RBAC with minimal perturbation. In: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 11–20. ACM (2008)
- [34] Vaidya, J., Atluri, V., Warner, J.: RoleMiner: Mining roles using subset enumeration. In: Proc. 13th ACM Conference on Computer and Communications Security (CCS). pp. 144–153. ACM (2006)
- [35] Vaidya, J., Atluri, V., Warner, J., Guo, Q.: Role engineering via prioritized subset enumeration. *IEEE Trans. Dependable Secur. Comput.* 7(3), 300–314 (2010), <http://dx.doi.org/10.1109/TDSC.2008.61>
- [36] Verde, N.V., Vaidya, J., Atluri, V., Colantonio, A.: Role engineering: From theory to practice. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY). pp. 181–192. ACM (2012), <http://doi.acm.org/10.1145/2133601.2133624>

- [37] eXtensible Access Control Markup Language (XACML), <http://www.oasis-open.org/committees/xacml/>
- [38] Xu, Z., Stoller, S.D.: Algorithms for mining meaningful roles. In: Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 57–66. ACM (2012)
- [39] 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>.
- [40] Xu, Z., Stoller, S.D.: Mining parameterized role-based policies. In: Proc. Third ACM Conference on Data and Application Security and Privacy (CODASPY). ACM (2013)
- [41] Zhang, D., Ramamohanarao, K., Ebringer, T.: Role engineering using graph optimisation. In: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies. pp. 139–144 (2007)
- [42] 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)
- [43] 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)

A Rule Quality Metric Based On Inductive Logic Programming

We review the theory quality metric used in Progol [25, 27], a well-known ILP system, and then describe our design of a rule quality metric based on it. Progol’s IPL algorithm works as follows. At the outermost level, Progol uses a loop that repeatedly generalizes an example to a hypothesized rule and then removes examples which are redundant relative to (i.e., covered by) the new rule, until no examples remain to be generalised. When generalizing an example, Progol uses a metric, called a *compression metric*, to guide construction of the hypotheses. When mining ABAC policies from operation logs, user-permissions tuples in $UP(L)$ are positive examples, and no negative examples are available. Thus, this corresponds to the case of learning from only positive data. When learning from only positive data, Progol’s compression metric $pcomp$ is defined as follows [26].

$$f_m(H) = c \times 2^{-|H|} (1 - g(H))^m \quad (2)$$

$$\begin{aligned} pcomp(H, E) &= \log_2 \frac{f_m(H)}{f_m(E)} \quad (3) \\ &= |E| - |H| - m(\log_2(1 - g(E)) \\ &\quad - \log_2(1 - g(H))) \\ &\approx |E| - |H| + m \log_2(1 - g(H)) \end{aligned}$$

where E is the set of positive examples, H is the entire theory (i.e., ABAC policy, in our context) being generated, including the part not generated yet, $m = |E|$, $|H|$ is the size of H , measured as the number of bits needed to encode H , and c is a constant chosen so that $\sum_{H \in \mathcal{H}} f_m(H) = 1$, where \mathcal{H} is the set of all candidate theories (note that f , like $pcomp$ is a function of H and E , since $m = |E|$, but we adopt Muggleton’s notation of $f_m(H)$ instead of using the more straightforward notation $f(H, E)$). Let X be the set of all possible well-formed examples (in our context, X is the set of all user-permission tuples). $g(H)$ (the “generality” of H) is the probability that an element of X , randomly selected following a uniform

distribution, satisfies H . f_m and $pcomp$ can be regarded as policy quality metrics. The term $2^{-|H|}$ in the definition of f_m causes policies with smaller size to have higher quality, and the term $(1 - g(H))^m$ causes policies with larger meaning to have higher quality. In [26], f_m is used to guide the search for rules. Using $pcomp$ to guide the search would have the same effect, because in the definition of $pcomp$ in equation (3), $f_m(E)$ is a constant, and $\log_2 \frac{f_m(H)}{f_m(E)}$ is a monotonic function of $f_m(H)$, so maximizing f_m is equivalent to maximizing $pcomp$.

A difficulty with using f_m to guide generation of a rule to add to a partly generated theory is that the entire theory H is not yet known. To overcome this difficulty, the quality of the entire theory is estimated by extrapolation. Let C_i denote the i 'th rule added to the theory, and let $H_i = \{C_1, \dots, C_i\}$. Let n denote the number of rules in the entire theory being generated (of course, n is not known until the algorithm terminates). When generating C_i , the policy quality $f_m(H_n)$ is estimated as follows.

$$c \times 2^{-\left(\frac{m}{p} \times |C_i|\right)} \times \left(1 - \frac{m}{p} * (g(H_i) - g(H_{i-1}))\right)^m \quad (4)$$

where p is the number of examples in E that are implied by C_i and not by H_{i-1} .

Now we describe how to modify our policy mining algorithm to use f_m as a rule quality metric. In the loop in the top-level pseudocode in Figure 1 that builds the set $Rules$ of candidate rules, rule quality is computed using equation 4 with $H_{i-1} = Rules$ and $C_i = \rho$. Specifically, in the definition of `generalizeRule` in 4, $Q_{rul}(\rho, uncovUP)$ is replaced with f_m computed using equation 4 with $H_{i-1} = Rules$ and $C_i = \rho$. This closely corresponds to the usage in Progol, although it is slightly different, because some of the candidate rules in $Rules$ will not be included in the final set of rules $Rules'$.

In the loop in the top-level pseudocode in Figure 1 that builds the set $Rules'$ of final rules, we take the same approach, except using $Rules'$ instead of $Rules$. Specifically, $Q_{rul}(\rho, uncovUP)$ is replaced with f_m computed using equation 4 with $H_{i-1} = Rules'$ and $C_i = \rho$.

To compute rule quality after building the set $Rules$ of candidate rules and before building the set $Rules'$ of final rules, the algorithm is modifying a set of rules, not extending a set of rules, so f_m can be evaluated using equation 2, with $Rules$ as an estimate of the entire policy H . Specifically, in the calls to `elimConjuncts` and `elimConstraints` from `simplifyRules`, $Q_{rul}(\rho'', UP_0)$ is replaced with f_m computed using equation 2 with $H = Rules \setminus \rho \cup \{\rho''\}$, and $Q_{rul}(\rho_{best}, UP_0)$ is replaced with f_m computed using equation 2 with $H = Rules \setminus \rho \cup \{\rho_{best}\}$.