Mining Attribute-based Access Control Policies

Zhongyuan Xu and Scott D. Stoller
Computer Science Department, Stony Brook University

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 an access control list (ACL) policy or role-based access control (RBAC) policy with accompanying attribute data. This paper presents an ABAC policy mining algorithm. To the best of our knowledge, it is the first ABAC policy mining algorithm. Our algorithm iterates over tuples in the given user-permission relation, 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. Our algorithm attempts to improve the policy by merging and simplifying candidate rules, and then it selects the highest-quality candidate rules for inclusion in the generated policy.

arXiv:1306.2401v4 [cs.CR] 7 Aug 2014

1 Introduction

Attribute-based access control (ABAC) provides a high level of flexibility that promotes security and information sharing [1]. ABAC also overcomes some of the problems associated with RBAC [2], notably role explosion [1], [3]. The benefits of ABAC led the Federal Chief Information Officer Council to call out ABAC as a recommended access control model in the Federal Identity Credential and Access Management Roadmap and Implementation Guidance, ver. 2.0 [1], [4].

Manual development of RBAC policies can be time-consuming and expensive [5]. Role mining algorithms promise to drastically reduce the cost, by partially automating the development of RBAC policies [5]. Role mining is an active research area and a currently relatively small (about $70 million) but rapidly growing commercial market segment [5]. Similarly, manual development of ABAC policies can be difficult [6] and expensive [1]. ABAC policy mining algorithms have potential to reduce the cost of ABAC policy development.

The main contribution of this paper is an algorithm for ABAC policy mining. Our algorithm is formulated to mine an ABAC policy from ACLs and attribute data. It can be used to mine an ABAC policy from an RBAC policy and attribute data, by expanding the RBAC policy into ACLs, adding a "role" attribute to the attribute data (to avoid information loss), and then applying our algorithm. At a high level, our algorithm works as follows. It iterates over tuples in the given user-permission relation, 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. We also developed an extension of the algorithm to identify suspected noise in the input.

Section 5 presents results from evaluating the algorithm on some relatively small but non-trivial hand-written sample policies and on synthetic (i.e., pseudo-randomly generated) policies. The general methodology is to start with an ABAC policy (including attribute data), generate an equivalent ACL policy from the ABAC policy, add noise (in some experiments) to the ACL policy and attribute data, run our algorithm on the resulting ACL policies and attribute data, and compare the mined ABAC policy with the original ABAC policy.

2 ABAC Policy Language

This section presents our ABAC policy language. We do not consider policy administration, since our goal is to mine a single ABAC policy from the current low-level policy. We present a specific concrete policy language, rather than a flexible framework, to simplify the exposition and evaluation of our policy mining algorithm, although our approach is general and can be adapted to other ABAC policy languages. Our ABAC policy language contains all of the common ABAC policy language constructs, except arithmetic inequalities and negation. Extending our algorithm to handle those constructs is future work. The policy language handled in this paper is already significantly more complex than policy languages handled in previous work on security policy mining.

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 $d_{\mathrm{u}}(u, a)$ is the value of attribute a for user

This material is based upon work supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grants CNS-0831298 and CNS-1421893. Submitted to IEEE TDSC. *©2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.*

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 $\bot$ used to indicate that an attribute's value is unknown. The set of possible values of multi-valued attributes is $\text{Val}_m = \text{Set}(\text{Val}_s \setminus \{\bot\}) \cup \bot$, 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 attribute a, or a set (interpreted as a disjunction) of possible values of a excluding $\bot$ (in other words, a subset of $\text{Val}_s \setminus \{\bot\}$ or $\text{Val}_m \setminus \{\bot\}$, depending on whether a is single-valued or multi-valued). We refer to the set $e(a)$ as the *conjunct* for attribute a. We say that expression e *uses* an attribute a if $e(a) \neq \top$. Let $\text{attr}(e)$ denote the set of attributes used by e. Let $\text{attr}_1(e)$ and $\text{attr}_m(e)$ denote the sets of single-valued and multi-valued attributes, respectively, 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 (and possibly other courses).

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 $a \supseteq\in v$ (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 $[\![e]\!]_U$, is the set of users in U that satisfy it: $[\![e]\!]_U = \{u \in U \mid u \models e\}$. 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).

In ABAC policy rules, constraints are used to 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 $d_u(u, \text{uAttr}(f)) \neq \bot$ and $d_r(u, \text{rAttr}(f)) \neq \bot$ 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 $\langle$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 $\langle U, R, \mathit{Op}, A_\mathrm{u}, A_\mathrm{r}, d_\mathrm{u}, d_\mathrm{r}, \mathit{Rules}\rangle$, 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 $[\![\rho]\!] = \{\langle u, r, o\rangle \in U \times R \times \mathit{Op} \mid \langle u, r, o\rangle \models \rho\}$. 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 $[\![\pi]\!] = \bigcup_{\rho \in \mathit{Rules}} [\![\rho]\!]$.

3 The ABAC Policy Mining Problem

An *access control list (ACL) policy* is a tuple $\langle U, R, \mathit{Op}, \mathit{UP}_0\rangle$, where U is a set of users, R is a set of resources, Op is a set of operations, and $\mathit{UP}_0 \subseteq U \times R \times \mathit{Op}$ is a user-permission relation, obtained from the union of the access control lists.

An ABAC policy π is *consistent* with an ACL policy $\langle U, P, \mathit{Op}, \mathit{UP}_0\rangle$ if they have the same sets of users, resource, and operations and $[\![\pi]\!] = \mathit{UP}_0$.

An ABAC policy consistent with a given ACL policy can be trivially constructed, by creating a separate rule corresponding to each user-permission tuple in the ACL policy, simply using uid and rid to identify the relevant user and resource. Of course, such an ABAC policy is as verbose and hard to manage as the original ACL policy. This observation forces us to ask: among ABAC policies semantically consistent with a given ACL policy π_0, which ones are preferable? We adopt two criteria.

One criterion is that policies that do not use the attributes uid and rid are preferable, because policies that use uid and rid are partly identity-based, not entirely attribute-based. Therefore, our definition of ABAC policy mining requires that these attributes are used only if necessary, i.e., only if every ABAC policy semantically consistent with π_0 contains rules that use them.

The other criterion is to maximize a policy quality metric. A *policy quality metric* is a function Q_pol from ABAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this is natural for metrics based on policy size. For generality, we parameterize the policy mining problem by the policy quality metric.

The *ABAC policy mining problem* is: given an ACL policy $\pi_0 = \langle U, R, \mathit{Op}, \mathit{UP}_0\rangle$, user attributes A_u, resource attributes A_r, user attribute data d_u, resource attribute data d_r, and a policy quality metric Q_pol, find a set Rules of rules such that the ABAC policy $\pi = \langle U, R, \mathit{Op}, A_\mathrm{u}, A_\mathrm{r}, d_\mathrm{u}, d_\mathrm{r}, \mathit{Rules}\rangle$ that (1) is consistent with π_0, (2) uses uid only when necessary, (3) uses rid only when necessary, and (4) has the best quality, according to Q_pol, among such policies.

The policy quality metric that our algorithm aims to optimize is *weighted structural complexity* (WSC) [7], a generalization of policy size. This is consistent with usability studies of access control rules, which conclude that more concise policies are more manageable [6]. Informally, the WSC of an ABAC policy is a weighted sum of the number of elements in the policy. Formally, the WSC of an ABAC policy π with rules Rules is $\mathrm{WSC}(\pi) = \mathrm{WSC}(\mathit{Rules})$, defined by

$$\begin{aligned}
\mathrm{WSC}(e) &= \sum_{a \in \mathrm{attr}_1(e)} |e(a)| + \sum_{a \in \mathrm{attr}_\mathrm{m}(e), s \in e(a)} |s| \\
\mathrm{WSC}(\langle e_\mathrm{u}, e_\mathrm{r}, O, c\rangle) &= w_1 \mathrm{WSC}(e_\mathrm{u}) + w_2 \mathrm{WSC}(e_\mathrm{r}) \\
&\quad + w_3 |O| + w_4 |c| \\
\mathrm{WSC}(\mathit{Rules}) &= \sum_{\rho \in \mathit{Rules}} \mathrm{WSC}(\rho),
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights.

Computational Complexity: We show that the ABAC policy mining problem is NP-hard, by reducing the Edge Role Mining Problem (Edge RMP) [8] to it. NP-hardness of Edge RMP follows from Theorem 1 in [7]. The basic idea of the reduction is that an Edge RMP instance I_R is translated into an ABAC policy mining problem instance I_A with uid and rid as the only attributes. Given a solution π_ABAC to problem instance I_A, the solution to I_R is constructed by interpreting each rule as a role. Details of the reduction appear in Section 8 in the Supplemental Material.

It is easy to show that a decision-problem version of ABAC policy mining is in NP. The decision-problem version asks whether there exists an ABAC policy that meets conditions (1)–(3) in the above definition of the ABAC policy mining problem and has WSC less than or equal to a given value.

4 Policy Mining Algorithm

Top-level pseudocode for our policy mining algorithm appears in Figure 1. It reflects the high-level structure described in Section 1. Functions called by the top-level pseudocode are described next. Function names hyperlink to pseudocode for the function, if it is included in the paper, otherwise to a description of the function. An example illustrating the processing of a user-permission tuple by our algorithm appears in Section 13 in the Supplemental Material. For efficiency, our algorithm incorporates heuristics and is not guaranteed to generate a policy with minimal WSC.

The function $\mathrm{addCandidateRule}(s_\mathrm{u}, s_r, s_\mathrm{o}, \mathit{cc}, \mathit{uncovUP}, \mathit{Rules})$ in Figure 2 first calls computeUAE to compute a user-attribute expression e_u that characterizes s_u, then calls and computeRAE to compute a resource-attribute expression e_r that characterizes s_r. It then calls $\mathrm{generalizeRule}(\rho, \mathit{cc}, \mathit{uncovUP}, \mathit{Rules})$ to generalize the 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 addCandidateRule are described next.

The function $\text{computeUAE}(s, U)$ computes a user-attribute expression e_{u} that characterizes the set s of users. The conjunct for each attribute a contains the values of a for users in s, unless one of those values is $\perp$, in which case a is unused (i.e., the conjunct for a is $\top$). Furthermore, the conjunct for uid is removed if the resulting attribute expression still characterizes s; this step is useful because policies that are not identity-based generalize better. Similarly, $\text{computeRAE}(s, R)$ computes a resource-attribute expression that characterizes the set s of resources. The attribute expressions returned by computeUAE and computeRAE might not be minimum-sized among expressions that characterize s: it is possible that some conjuncts can be removed. We defer minimization of the attribute expressions until after the call to generalizeRule (described below), because minimizing them before that would reduce opportunities to find relations between values of user attributes and resource attributes in generalizeRule.

The function $\text{candidateConstraint}(r, u)$ returns a set containing all the atomic constraints that hold between resource r and user u. Pseudocode for $\text{candidateConstraint}$ is straightforward and omitted.

A rule ρ' is *valid* if $[\![\rho']\!] \subseteq UP_0$.

The function $\text{generalizeRule}(\rho, cc, \mathit{uncovUP}, \mathit{Rules})$ in Figure 3 attempts to generalize rule ρ by adding some of the atomic constraints f in cc to ρ and eliminating the conjuncts of the user attribute expression and the resource attribute expression corresponding to the attributes used in f, i.e., mapping those attributes to $\top$. If the resulting rule is invalid, the function attempts a more conservative generalization by eliminating only one of those conjuncts, keeping the other. 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.

If there are no valid generalizations of ρ, $\text{generalizeRule}(\rho, cc, \mathit{uncovUP}, \mathit{Rules})$ returns ρ. If there is a valid generalization of ρ, $\text{generalizeRule}(\rho, cc, \mathit{uncovUP}, \mathit{Rules})$ 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. Based on our primary goal of minimizing the generated policy's WSC, and a secondary preference for rules with more constraints, we define

$$Q_{\text{rul}}(\rho, UP) = \langle |\, [\![\rho]\!] \cap UP| / \text{WSC}(\rho),\ |\text{con}(\rho)| \rangle.$$

The secondary preference for more constraints is a heuristic, based on the observation that rules with more constraints tend to be more general than other rules with the same $|\,[\![\rho]\!] \cap UP|/\text{WSC}(\rho)$ (such rules typically have more conjuncts) and hence lead to lower WSC. In generalizeRule, $\mathit{uncovUP}$ 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.

The function $\text{mergeRules}(\mathit{Rules})$ in Figure 4 attempts to reduce the WSC 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 $[\![\rho]\!] \subseteq [\![\rho']\!]$. 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 the resulting rule ρ_{merge} is valid, ρ_{merge} is added to Rules, and ρ_1 and ρ_2 and any other rules that are now redundant are removed from Rules. $\text{mergeRules}(\mathit{Rules})$ updates its argument Rules in place, and it returns a Boolean indicating whether any rules were merged.

```
    // Rules is the set of candidate rules
1:  Rules = ∅
    // uncovUP contains user-permission tuples in UP_0
    // that are not covered by Rules
2:  uncovUP = UP_0.copy()
3:  while ¬uncovUP.isEmpty()
       // Select an uncovered user-permission tuple.
4:     ⟨u, r, o⟩ = some tuple in uncovUP
5:     cc = candidateConstraint(r, u)
       // s_u contains users with permission ⟨r, o⟩ and
       // that have the same candidate constraint for r as u
6:     s_u = {u' ∈ U | ⟨u', r, o⟩ ∈ UP_0
7:                  ∧ candidateConstraint(r, u') = cc}
8:     addCandidateRule(s_u, {r}, {o}, cc, uncovUP, Rules)
       // s_o is set of operations that u can apply to r
9:     s_o = {o' ∈ Op | ⟨u, r, o'⟩ ∈ UP_0}
10:    addCandidateRule({u}, {r}, s_o, cc, uncovUP, Rules)
11: end while
    // Repeatedly merge and simplify rules, until
    // this has no effect
12: mergeRules(Rules)
13: while simplifyRules(Rules) && mergeRules(Rules)
14:    skip
15: end while
    // Select high quality rules into final result Rules'.
16: Rules' = ∅
17: Repeatedly select the highest quality rules from
    Rules to Rules' until Σ_{ρ∈Rules'} [[ρ]] = UP_0,
    using UP_0 \ [[Rules']] as second argument to Q_rul
18: return Rules'
```

Fig. 1. Policy mining algorithm.

```
function addCandidateRule(s_u, s_r, s_o, cc, uncovUP, Rules)
  // Construct a rule ρ that covers user-permission
  // tuples {⟨u, r, o⟩ | u ∈ s_u ∧ r ∈ s_r ∧ o ∈ s_o}.
1: e_u = computeUAE(s_u, U)
2: e_r = computeRAE(s_r, R)
3: ρ = ⟨e_u, e_r, s_o, ∅⟩
4: ρ′ = generalizeRule(ρ, cc, uncovUP, Rules)
5: Rules.add(ρ′)
6: uncovUP.removeAll([[ρ′]])
```

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

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 the following ways. (1) It eliminates sets that are supersets of other sets in conjuncts for multi-valued user attributes. The $\supseteq$-based semantics for such conjuncts implies that this does not change the meaning of the conjunct. For example, a conjunct $\{\{a\},\{a,b\}\}$ is simplified to $\{\{a\}\}$. (2) It eliminates elements from sets in conjuncts for multi-valued user attributes when this preserves validity of the rule; note that this might increase but cannot decrease the meaning of a rule. For example, if every user whose specialties include a also have specialty b, and a rule contains the conjunct $\{\{a,b\}\}$ for the specialties attribute, then b will be eliminated from that conjunct. (3) It eliminates conjuncts from a rule when this preserves validity of the rule. Since removing one conjunct might prevent removal of another conjunct, it searches for the set of conjuncts to remove that maximizes the quality of the resulting rule, while preserving validity. The user can specify a set of *unremovable attributes*, i.e., attributes for which simplifyRules should not try to eliminate the conjunct, because eliminating it would increase 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. Our experience suggests that appropriate unremovable attributes can be identified based on the obvious importance of some attributes and by examination of the policy generated without specification of unremovable attributes. (4) It eliminates atomic constraints from a rule when this preserves validity of the rule. It searches for the set of atomic constraints to remove that maximizes the quality of the resulting rule, while preserving validity. (5) It eliminates overlapping values between rules. Specifically, a value v in the conjunct for a user attribute a in a rule ρ is removed if there is another rule ρ' in the policy such that (*i*) $\mathrm{attr}(\mathrm{uae}(\rho')) \subseteq \mathrm{attr}(\mathrm{uae}(\rho))$ and $\mathrm{attr}(\mathrm{rae}(\rho')) \subseteq \mathrm{attr}(\mathrm{rae}(\rho))$, (*ii*) the conjunct of $\mathrm{uae}(\rho')$ for a contains v, (*iii*) each conjunct of $\mathrm{uae}(\rho')$ or $\mathrm{rae}(\rho')$ other than the conjunct for a is either $\top$ or a superset of the

```
function generalizeRule(ρ, cc, uncovUP, Rules)
  // ρ_best is highest-quality generalization of ρ
1: ρ_best = ρ
  // cc′ contains formulas from cc that lead to valid
  // generalizations of ρ.
2: cc′ = new Vector()
3: // gen[i] is a generalization of ρ using cc′[i]
4: gen = new Vector()
  // find formulas in cc that lead to valid
  // generalizations of ρ.
5: for f in cc
     // try to generalize ρ by adding f and elimi-
     // nating conjuncts for both attributes used in f.
6:   ρ′ = ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ)[rAttr(f) ↦ ⊤],
7:         ops(ρ), con(ρ) ∪ {f}⟩
     // check if [[ρ′]] is a valid rule
8:   if [[ρ′]] ⊆ UP_0
9:     cc′.add(f)
10:    gen.add(ρ′)
11:  else
       // try to generalize ρ by adding f and elimi-
       // nating conjunct for one user attribute used in f
12:    ρ′ = ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ),
13:          ops(ρ), con(ρ) ∪ {f}⟩
14:    if [[ρ′]] ⊆ UP_0
15:      cc′.add(f)
16:      gen.add(ρ′)
17:    else
     // try to generalize ρ by adding f and elimi-
     // nating conjunct for one resource attribute used in f.
18:      ρ′ = ⟨uae(ρ), rae(ρ)[rAttr(f) ↦ ⊤],
19:            ops(ρ), con(ρ) ∪ {f}⟩
20:      if [[ρ′]] ⊆ UP_0
21:        cc′.add(f)
22:        gen.add(ρ′)
23:      end if
24:    end if
25:  end if
26: end for
27: for i = 1 to cc′.length
28:   // try to further generalize gen[i]
29:   ρ″ = generalizeRule(gen[i], cc′[i+1 ..], uncovUP,
30:                       Rules)
31:   if Q_rul(ρ″, uncovUP) > Q_rul(ρ_best, uncovUP)
32:     ρ_best = ρ″
33:   end if
34: end for
35: return ρ_best
```

Fig. 3. 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.

```
    function mergeRules(Rules)
1:  // Remove redundant rules
2:  rdtRules = {ρ ∈ Rules | ∃ ρ' ∈ Rules \ {ρ}. [[ρ]] ⊆ [[ρ']]}
3:  Rules.removeAll(rdtRules)
4:  // Merge rules
5:  workSet = {(ρ1, ρ2) | ρ1 ∈ Rules ∧ ρ2 ∈ Rules
                        ∧ ρ1 ≠ ρ2 ∧ con(ρ1) = con(ρ2)}
6:  while not(workSet.empty())
      // Remove an arbitrary element of the workset
7:    (ρ1, ρ2) = workSet.remove()
8:    ρmerge = ⟨uae(ρ1) ∪ uae(ρ2), rae(ρ1) ∪ rae(ρ2),
                ops(ρ1) ∪ ops(ρ2), con(ρ1)⟩
9:    if [[ρmerge]] ⊆ UP0
        // The merged rule is valid. Add it to Rules,
        // and remove rules that became redundant.
10:     rdtRules = {ρ ∈ Rules | [[ρ]] ⊆ [[ρmerge]]}
11:     Rules.removeAll(rdtRules)
12:     workSet.removeAll({(ρ1, ρ2) ∈ workSet |
                  ρ1 ∈ rdtRules ∨ ρ2 ∈ rdtRules})
13:     workSet.addAll({(ρmerge, ρ) | ρ ∈ Rules
                        ∧ con(ρ) = con(ρmerge)})
14:     Rules.add(ρmerge)
15:   end if
16: end while
17: return true if any rules were merged
```

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

corresponding conjunct of ρ, and (*iv*) $\mathrm{con}(\rho') \subseteq \mathrm{con}(\rho)$. The condition for removal of a value in the conjunct for a resource attribute is analogous. If a conjunct of $\mathrm{uae}(\rho)$ or $\mathrm{rae}(\rho)$ becomes empty, ρ is removed from the policy. For example, if a policy contains the rules $\langle \mathrm{dept} \in \{d_1, d_2\} \wedge \mathrm{position} = p_1, \mathrm{type} = t_1, \mathrm{read}, \mathrm{dept} = \mathrm{dept}\rangle$ and $\langle \mathrm{dept} \in \{d_1\} \wedge \mathrm{position} = p_1, \mathrm{type} \in \{t_1, t_2\}, \mathrm{read}, \mathrm{dept} = \mathrm{dept}\rangle$, then d_1 is eliminated from the former rule. (6) It eliminates overlapping operations between rules. The details are similar to those for elimination of overlapping values between rules. For example, if a policy contains the rules $\langle \mathrm{dept} = d_1, \mathrm{type} = t_1, \mathrm{read}, \mathrm{dept} = \mathrm{dept}\rangle$ and $\langle \mathrm{dept} = d_1 \wedge \mathrm{position} = p_1, \mathrm{type} = t_1, \{\mathrm{read}, \mathrm{write}\}, \mathrm{dept} = \mathrm{dept}\rangle$, then read is eliminated from the latter rule.

Asymptotic Running Time: The algorithm's overall running time is worst-case cubic in $|UP_0|$. A detailed analysis of the asymptotic running time appears in Section 9 in the Supplemental Material. In the experiments with sample policies and synthetic policies described in Section 5, the observed running time is roughly quadratic and roughly linear, respectively, in $|UP_0|$.

Attribute Selection: Attribute data may contain attributes irrelevant to access control. This potentially hurts the effectiveness and performance of policy mining algorithms [9], [10]. Therefore, before applying our algorithm to a dataset that might contain irrelevant attributes, it is advisable to use the method in [9] or [11] to determine the relevance of each attribute to the user-permission assignment and then eliminate attributes with low relevance.

Processing Order: The order in which tuples and rules are processed can affect the mined policy. The order in which our algorithm processes tuples and rules is described in Section 10 in the Supplemental Material.

Optimizations: Our implementation incorporates a few optimizations not reflected in the pseudocode but described in Section 11 in the Supplemental Material. The most novel optimization is that rules are merged (by calling mergeRules) periodically, not only after all of UP_0 has been covered. This is beneficial because merging sometimes has the side-effect of generalization, which causes more user-permission tuples to be covered without explicitly considering them as seeds.

4.1 Noise Detection

In practice, the given user-permission relation often contains noise, consisting of over-assignments and under-assignments. An *over-assignment* is when a permission is inappropriately granted to a user. An *under-assignment* is when a user lacks a permission that he or she should be granted. Noise incurs security risks and significant IT support effort [11]. This section describes extensions of our algorithm to handle noise. The extended algorithm detects and reports suspected noise and generates an ABAC policy that is consistent with its notion of the correct user-permission relation (i.e., with the suspected noise removed). The user should examine the suspected noise and decide which parts of it are actual noise (i.e., errors in the user-permission relation). If all of it is actual noise, then the policy already generated is the desired one; otherwise, the user should remove the parts that are actual noise from the user-permission relation to obtain a correct user-permission relation and then run the algorithm without the noise detection extension on it to generate the desired ABAC policy.

Over-assignments are often the result of incomplete revocation of old permissions when users change job functions [11]. Therefore, over-assignments usually cannot be captured concisely using rules with attribute expressions that refer to the current attribute information, so a candidate rule constructed from a user-permission tuple that is an over-assignment is less likely to be generalized and merged with other rules, and that candidate rule will end up as a low-quality rule in the generated policy. So, to detect over-assignments, we introduce a rule quality threshold τ. The rule quality metric used here is the first component of the metric used in the loop in Figure 1 that constructs $Rules'$; thus, τ is a threshold on the value of $Q_{\mathrm{rul}}(\rho, uncovUP)$, and the rules with quality less than or equal to τ form a suffix of the sequence of rules added

to $Rules'$. The extended algorithm reports as suspected over-assignments the user-permission tuples covered in $Rules'$ only by rules with quality less than or equal to τ, and then it removes rules with quality less than or equal to τ from $Rules'$. Adjustment of τ is guided by the user. For example, the user might guess a percentage of over-assignments (e.g., 3%) based on experience, and let the system adjust τ until the number of reported over-assignments is that percentage of $|UP_0|$.

To detect under-assignments, we look for rules that are almost valid, i.e., rules that would be valid if a relatively small number of tuples were added to UP_0. A parameter α quantifies the notion of "relatively small". A rule is α *almost valid* if the fraction of invalid user-permission tuples in $[\![\rho]\!]$ is at most α, i.e., $|\,[\![\rho]\!] \setminus UP_0| \div |\,[\![\rho]\!]\,| \leq \alpha$. In places where the policy mining algorithm checks whether a rule is valid, if the rule is α almost valid, the algorithm treats it as if it were valid. The extended algorithm reports $\bigcup_{\rho \in Rules'} [\![\rho]\!] \setminus UP_0$ as the set of suspected under-assignments, and (as usual) it returns $Rules'$ as the generated policy. Adjustment of α is guided by the user, similarly as for the over-assignment threshold τ.

5 Evaluation

The general methodology used for evaluation is described in Section 1. We applied this methodology to sample policies and synthetic policies. Evaluation on policies (including attribute data) from real organizations would be ideal, but we are not aware of any suitable and publicly available policies from real organizations. Therefore, we developed sample policies that, although not based directly on specific real-world case studies, are intended to be similar to policies that might be found in the application domains for which they are named. The sample policies are relatively small and intended to resemble interesting core parts of full-scale policies in those application domains. Despite their modest size, they are a significant test of the effectiveness of our algorithm, because they express non-trivial policies and exercise all features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The synthetic policies are used primarily to assess the behavior of the algorithm as a function of parameters controlling specific structural characteristics of the policies.

We implemented our policy mining algorithm in Java and ran experiments on a laptop with a 2.5 GHz Intel Core i5 CPU. All of the code and data is available at http://www.cs.sunysb.edu/~stoller/. In our experiments, the weights w_i in the definition of WSC equal 1.

5.1 Evaluation on Sample Policies

We developed four sample policies, each consisting of rules and a manually written attribute dataset containing a small number of instances of each type of user and resource. We also generated synthetic attribute datasets for each sample policy. The sample policies are described very briefly in this section. Details of the sample policies, including all policy rules, some illustrative manually written attribute data, and a more detailed description of the synthetic attribute data generation algorithm appear in Section 12 in the Supplemental Material.

Figure 5 provides information about their size. Although the sample policies are relatively small when measured by a coarse metric such as number of rules, they are complex, because each rule has a lot of structure. For example, the number of well-formed rules built using the attributes and constants in each policy and that satisfy the strictest syntactic size limits satisfied by rules in the sample policies (at most one conjunct in each UAE, at most two conjuncts in each RAE, at most two atomic constraints in each constraint, at most one atomic value in each UAE conjunct, at most two atomic values in each RAE conjunct, etc.) is more than 10^{12} for the sample policies with manually written attribute data and is much higher for the sample policies with synthetic attribute data and the synthetic policies.

In summary, our algorithm is very effective for all three sample policies: there are only small differences between the original and mined policies if no attributes are declared unremovable, and the original and mined policies are identical if the resource-type attribute is declared unremovable.

University Sample Policy: Our university sample policy controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=transcript" is replaced with the constraint "department=department" in one rule. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Health Care Sample Policy: Our health care sample policy 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). If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=HRitem" is eliminated from four rules; that conjunct is unnecessary, because those rules also contain a conjunct for the "topic" attribute, and the "topic" attribute is used only for resources with type=HRitem. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Project Management Sample Policy: Our project management sample policy controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=task" is eliminated from three rules; the

Policy	$\|Rules\|$	$\|A_u\|$	$\|A_r\|$	$\|Op\|$	Type	N	$\|U\|$	$\|R\|$	$\|Val_s\|$	$\|UP\|$	$\|\widehat{[\![\rho]\!]}\|$
university	10	6	5	9	man	2	22	34	76	168	19
					syn	10	479	997	1651	8374	837
					syn	20	920	1918	3166	24077	2408
health care	9	6	7	3	man	2	21	16	55	51	6.7
					syn	10	200	720	1386	1532	195
					syn	20	400	1440	2758	3098	393
project mgmt	11	8	6	7	man	2	19	40	77	189	19
					syn	10	100	200	543	960	96
					syn	20	200	400	1064	1920	193

Fig. 5. Sizes of the sample policies. "Type" indicates whether the attribute data in the policy is manually written ("man") or synthetic ("syn"). N is the number of departments for the university and project management sample policies, and the number of wards for the health care sample policy. $|\widehat{[\![\rho]\!]}|$ is the average number of user-permission tuples that satisfy each rule. An empty cell indicates the same value as the cell above it.

explanation is similar to the above explanation for the health care sample policy. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Running Time on Synthetic Attribute Data: We generated a series of pseudorandom synthetic attribute datasets for the sample policies, parameterized by a number N, which is the number of departments for the university and project management sample policies, and the number of wards for the health care sample policy. 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. Figure 5 contains information about the sizes of the policies with synthetic attribute data, for selected values of N. Policies for the largest shown value of N are generated as described in Section 12 in the Supplemental Material; policies for smaller values of N are prefixes of them. Each row contains the average over 20 synthetic policies with the specified N. For all sizes of synthetic attribute data, the mined policies are the same as with the manually generated attribute data. This reflects that larger attribute datasets are not necessarily harder to mine from, if they represent more instances of the same rules; the complexity is primarily in the structure of the rules. Figure 6 shows the algorithm's running time as a function of N. Each data point is an average of the running times on 20 policies with synthetic attribute data. Error bars (too small to see in most cases) show 95% confidence intervals using Student's t-distribution. The running time is a roughly quadratic function of N for all three sample policies, with different constant factors. Different constant factors are expected, because policies are very complex structures, and N captures only one aspect of the size and difficulty of the policy mining problem instance. For example, the constant factors are larger for the university sample policy mainly because it has larger $|UP|$, as a function of N, than the other sample policies. For example, Figure 5 shows that $|UP|$ for the university sample policy with $N = 10$ is larger than $|UP|$ for the other sample policies with $N = 20$.

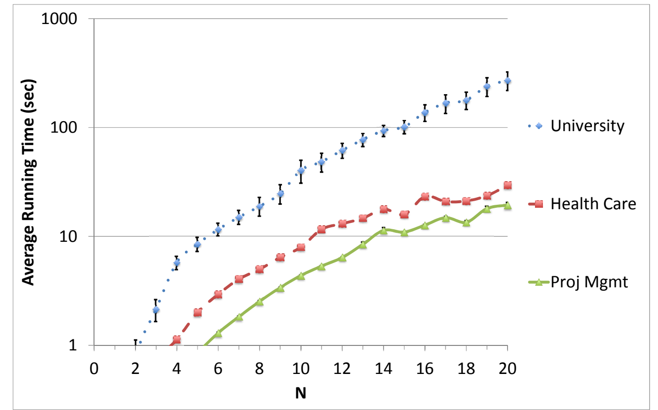

Fig. 6. Running time (log scale) of the algorithm on synthetic attribute datasets for sample policies. The horizontal axis is N_{dept} for university and project management sample policies and N_{ward} for health care sample policy.

Benefit of Periodic Rule Merging Optimization: It is not obvious *a priori* whether the savings from periodic merging of rules outweighs the cost. In fact, the net benefit grows with policy size. For example, for the university policy with synthetic attribute data, this optimization provides a speedup of (67 sec)/(40 sec) = 1.7 for $N_{\mathrm{dept}} = 10$ and a speedup of (1012 sec)/(102 sec) = 9.9 for $N_{\mathrm{dept}} = 15$.

5.2 Evaluation on Synthetic Policies

We also evaluated our algorithm on synthetic ABAC policies. On the positive side, synthetic policies can be generated in all sizes and with varying structural characteristics. On the other hand, even though our synthesis algorithm is designed to generate policies with some realistic characteristics, the effectiveness and performance of our algorithm on synthetic policies might not be representative of its effectiveness and performance on real policies. For experiments with synthetic policies, we

compare the syntactic similarity and WSC of the synthetic ABAC policy and the mined ABAC policy. Syntactic similarity of policies measures the syntactic similarity of rules in the policies. It ranges from 0 (completely different) to 1 (identical). The detailed definition of syntactic similarity is in Section 14 in the Supplemental Material. We do not expect high syntactic similarity between the synthetic and mined ABAC policies, because synthetic policies tend to be unnecessarily complicated, and mined policies tend to be more concise. Thus, we consider the policy mining algorithm to be effective if the mined ABAC policy $Rules_{\rm mined}$ is simpler (i.e., has lower WSC) than the original synthetic ABAC policy $Rules_{\rm syn}$. We compare them using the *compression factor*, defined as $\mathrm{WSC}(Rules_{\rm syn})/\mathrm{WSC}(Rules_{\rm mined})$. Thus, a compression factor above 1 is good, and larger is better.

Synthetic Policy Generation: Our 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. Our synthesis algorithm takes $N_{\rm rule}$, the desired number of rules, $N_{\rm cnj}^{\rm min}$, the minimum number of conjuncts in each attribute expression, and $N_{\rm cns}^{\rm min}$, the minimum number of constraints in each rule, as inputs. The numbers of users and resources are not specified directly but are proportional to the number of rules, since our algorithm generates new users and resources to satisfy each generated rule, as sketched below. Rule generation is based on several statistical distributions, which are either based loosely on our sample policies or assumed to have a simple functional form (e.g., uniform distribution or Zipf distribution). For example, the distribution of the number of conjuncts in each attribute expression is based loosely on our sample policies and ranges from $N_{\rm cnj}^{\rm min}$ to $N_{\rm cnj}^{\rm min}+3$, the distribution of the number of atomic constraints in each constraint is based loosely on our sample policies and ranges from $N_{\rm cns}^{\rm min}$ to $N_{\rm cns}^{\rm min}+2$, and the distribution of attributes in attribute expressions is assumed to be uniform (i.e., each attribute is equally likely to be selected for use in each conjunct).

The numbers of user attributes and resource attributes are fixed at $N_{\rm attr}=8$ (this is the maximum number of attributes relevant to access control for the datasets presented in [12]). Our synthesis algorithm adopts a simple type system, with 7 types, and with at least one user attribute and one resource attribute of each type. For each type t, the cardinality $c(t)$ is selected from a uniform distribution on the interval $[2, 10N_{\rm rule}+2]$, the target ratio between the frequencies of the most and least frequent values of type t is chosen to be 1, 10, or 100 with probability 0.2 0.7, and 0.1, respectively, and a skew $s(t)$ is computed so that the Zipf distribution with cardinality $c(t)$ and skew $s(t)$ has that frequency ratio. When assigning a value to an attribute of type t, the value is selected from the Zipf distribution with cardinality $c(t)$ and skew $s(t)$. Types are also used when generating constraints: constraints relate attributes with the same type.

For each rule ρ, our algorithm ensures that there are at least $N_{\rm urp}=16$ user-resource pairs $\langle u,r\rangle$ such that $\langle u,r,o\rangle\models\rho$ for some operation o. The algorithm first checks how many pairs of an existing user and an existing resource (which were generated for previous rules) satisfy ρ or can be made to satisfy ρ by appropriate choice of values for attributes with unknown values (i.e., $\bot$). If the count is less than $N_{\rm urp}$, the algorithm generates additional users and resources that together satisfy ρ. With the resulting modest number of users and resources, some conjuncts in the UAE and RAE are likely to be unnecessary (i.e., eliminating them does not grant additional permissions to any existing user). In a real policy with sufficiently large numbers of users and resources, all conjuncts are likely be to necessary. To emulate this situation with a modest number of users, for each rule ρ, for each conjunct $e_u(a_u)$ in the UAE e_u in ρ, the algorithm generates a user u' by copying an existing user u that (together with some resource) satisfies ρ and then changing $d_{\rm u}(u',a_u)$ to some value not in $e_u(a_u)$. Similarly, the algorithm adds resources to increase the chance that conjuncts in resource attribute expressions are necessary, and it adds users and resources to increase the chance that constraints are necessary. The algorithm initially assigns values only to the attributes needed to ensure that a user or resource satisfies the rule under consideration. To make the attribute data more realistic, a final step of the algorithm assigns values to additional attributes until the fraction of attribute values equal to $\bot$ reaches a target fraction $\nu_\bot=0.1$.

Results for Varying Number of Conjuncts: To explore the effect of varying the number of conjuncts, we generated synthetic policies with $N_{\rm rule}$ ranging from 10 to 50 in steps of 20, with $N_{\rm cnj}^{\rm min}$ ranging from 4 to 0, and with $N_{\rm cns}^{\rm min}=0$. For each value of $N_{\rm rule}$, synthetic policies with smaller $N_{\rm cnj}^{\rm min}$ are obtained by removing conjuncts from synthetic policies with larger $N_{\rm cnj}^{\rm min}$. For each combination of parameter values (in these experiments and the experiments with varying number of constraints and varying overlap between rules), we generate 50 synthetic policies and average the results. Some experimental results appear in Figure 7. For each value of $N_{\rm rule}$, as the number of conjuncts decreases, $|UP|$ increases (because the numbers of users and resources satisfying each rule increase), the syntactic similarity increases (because as there are fewer conjuncts in each rule in the synthetic policy, it is more likely that the remaining conjuncts are important and will also appear in the mined policy), and the compression factor decreases (because as the policies get more similar, the compression factor must get closer to 1). For example, for $N_{\rm rule}=50$, as $N_{\rm cnj}^{\rm min}$ decreases from 4 to 0, average $|UP|$ increases from 1975 to 11969, average syntactic similarity increases from 0.62 to 0.75, and average compression factor decreases from 1.75 to 0.84. The figure also shows the density of the policies, where the density of a policy is defined as

$|UP| \div (|U| \times |P|)$, where the set of granted permissions is $P = \bigcup_{\langle u,r,o\rangle \in UP} \{\langle r, o\rangle\}$. The average densities all fall within the range of densities seen in the 9 real-world datasets shown in [13, Table 1], namely, 0.003 to 0.19. Density is a decreasing function of $N_{\rm rule}$, because $|UP|$, $|U|$, and $|P|$ each grow roughly linearly as functions of $N_{\rm rule}$. The standard deviations of some quantities are relatively large in some cases, but, as the relatively small confidence intervals indicate, this is due to the intrinsic variability of the synthetic policies generated by our algorithm, not due to insufficient samples.

Results for Varying Number of Constraints: To explore the effect of varying the number of constraints, we generated synthetic policies with $N_{\rm rule}$ ranging from 10 to 50 in steps of 20, with $N_{\rm cns}^{\rm min}$ ranging from 2 to 0, and with $N_{\rm cnj}^{\rm min} = 0$. For each value of $N_{\rm rule}$, policies with smaller $N_{\rm cns}^{\rm min}$ are obtained by removing constraints from synthetic policies with larger $N_{\rm cns}^{\rm min}$. Some experimental results appear in Figure 8. For each value of $N_{\rm rule}$, as the number of constraints decreases, $|UP|$ increases (because the numbers of users and resources satisfying each rule increase), syntactic similarity decreases (because our algorithm gives preference to constraints over conjuncts, so when $N_{\rm cns}^{\rm min}$ is small, the mined policy tends to have more constraints and fewer conjuncts than the synthetic policy), and the compression factor decreases (because the additional constraints in the mined policy cause each rule in the mined policy to cover fewer user-permission tuples on average, increasing the number of rules and hence the WSC). For example, for $N_{\rm rule} = 50$, as $N_{\rm cns}^{\rm min}$ decreases from 2 to 0, average $|UP|$ increases from 3560 to 26472, average syntactic similarity decreases slightly from 0.67 to 0.64, and average compression factor decreases from 1.29 to 0.96.

Results for Varying Overlap Between Rules: We also explored the effect of varying overlap between rules, to test our conjecture that policies with more overlap between rules are harder to reconstruct through policy mining. The *overlap* between rules ρ_1 and ρ_2 is $[\![\rho_1]\!] \cap [\![\rho_2]\!]$. To increase the average overlap between pairs of rules in a synthetic policy, we extended the policy generation algorithm so that, after generating each rule ρ, with probability $P_{\rm over}$ the algorithm generates another rule ρ' obtained from ρ by randomly removing one conjunct from ${\rm uae}(\rho)$ and adding one conjunct (generated in the usual way) to ${\rm rae}(\rho)$; typically, ρ and ρ' have a significant amount of overlap. We also add users and resources that together satisfy ρ', so that $[\![\rho']\!] \not\subseteq [\![\rho]\!]$, otherwise ρ' is redundant. This construction is based on a pattern that occurs a few times in our sample policies. We generated synthetic policies with 30 rules, using the extended algorithm described above. For each value of $N_{\rm rule}$, we generated synthetic policies with $P_{\rm over}$ ranging from 0 to 1 in steps of 0.25, and with $N_{\rm cnj}^{\rm min}$ = 2 and $N_{\rm cns}^{\rm min}$ = 0. Some experimental results appear in Figure 9. For each value of $N_{\rm rule}$, as $P_{\rm over}$ increases, the syntactic similarity decreases (because our algorithm effectively removes overlap, i.e., produces policies with relatively little overlap), and the compression factor increases (because removal of more overlap makes the mined policy more concise). For example, for $N_{\rm rule} = 50$, as $P_{\rm over}$ increases from 0 to 1, the syntactic similarity decreases slightly from 0.74 to 0.71, and the compression factor increases from 1.16 to 1.23.

5.3 Generalization

A potential concern with optimization-based policy mining algorithms is that the mined policies might overfit the given data and hence not be robust, i.e., not generalize well, in the sense that the policy requires modifications to accommodate new users. To evaluate how well policies generated by our algorithm generalize, we applied the following methodology, based on [9]. The inputs to the methodology are an ABAC policy mining algorithm, an ABAC policy π, and a fraction f (informally, f is the fraction of the data used for training); the output is a fraction e called the *generalization error* of the policy mining algorithm on policy π for fraction f. Given a set U' of users and a policy π, the associated resources for U' are the resources r such that π grants some user in U' some permission on r. To compute the generalization error, repeat the following procedure 10 times and average the results: randomly select a subset U' of the user set U of π with $|U'|/|U| = f$, randomly select a subset R' of the associated resources for U' with $|R'|/|R| = f$, generate an ACL policy $\pi_{\rm ACL}$ containing only the permissions for users in U' for resources in R', apply the policy mining algorithm to $\pi_{\rm ACL}$ with the attribute data to generate an ABAC policy $\pi_{\rm gen}$, compute the generalization error as the fraction of incorrectly assigned permissions for users not in U' and resources not in R', i.e., as $|S \ominus S'|/|S|$, where $S = \{\langle u,r,o\rangle \in [\![\pi]\!] \mid u \in U \setminus U' \wedge r \in R \setminus R'\}$, $S' = \{\langle u,r,o\rangle \in [\![\pi']\!] \mid u \in U \setminus U' \wedge r \in R \setminus R'\}$, and $\ominus$ is symmetric set difference.

We measured generalization error for f from 0.1 to 0.5 in steps of 0.05 for the university (with $N_{\rm dept} = 40$), health care (with $N_{\rm ward} = 40$), and project management (with $N_{\rm dept} = 40$) sample policies. For the university and health care sample policies, the generalization error is zero in all these cases. For the project management sample policy, the generalization error is 0.11 at $f = 0.1$, drops roughly linearly to zero at $f = 0.35$, and remains zero thereafter. There are no other existing ABAC policy mining algorithms, so a direct comparison of the generalization results from our algorithm with generalization results from algorithms based on other approaches, e.g., probabilistic models, is not currently possible. Nevertheless, these results are promising and suggest that policies generated by our algorithm generalize reasonably well.

5.4 Noise

Permission Noise: To evaluate the effectiveness of our noise detection techniques in the presence of permission noise, we started with an ABAC policy, generated

N_{rule}	$N_{\text{cnj}}^{\min}$	$\|U\|$		$\|R\|$		$\|UP\|$		$\|\widehat{[\![\rho]\!]}\|$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
10	4	206	16	62	5.7	401	90	35	9.2	.069	.01	.63	0.04	0.011	1.79	.21	.060	0.30	0.31	0.09
	2					620	116	136	29	.076	.01	.69	0.03	0.009	1.55	.18	.051	0.47	0.19	0.05
	0					1025	222	298	62	.081	.01	.78	0.05	0.014	1.12	.16	.045	1.02	0.42	0.12
50	4	1008	38	318	15	1975	282	36	5.3	.014	.001	.62	0.02	0.006	1.75	.08	.023	4.86	1.71	0.49
	2					3314	526	144	20	.015	.001	.68	0.02	0.006	1.52	.08	.023	11.78	3.41	0.97
	0					11969	5192	438	136	.025	.007	.75	0.03	0.009	0.84	.18	.051	58.88	18.7	5.31

Fig. 7. Experimental results for synthetic policies with varying $N_{\text{cnj}}^{\min}$. "Synt. Sim." is syntactic similarity. "Compression" is the compression factor. μ is mean, σ is standard deviation, and CI is half-width of 95% confidence interval using Student's t-distribution. An empty cell indicates the same value as the cell above it.

N_{rule}	$N_{\text{cns}}^{\min}$	$\|U\|$		$\|R\|$		$\|UP\|$		$\|\widehat{[\![\rho]\!]}\|$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
10	2	172	19	54	6.5	529	162	49	15	.084	.020	.72	.04	.011	1.50	.14	.040	0.28	0.17	0.05
	1					679	174	105	32	.093	.018	.72	.05	.014	1.43	.18	.051	0.36	0.17	0.05
	0					917	325	172	57	.110	.034	.70	.05	.014	1.30	.20	.057	0.49	0.20	0.06
50	2	781	51	276	16	3560	596	61	9.9	.020	.003	.67	.02	.006	1.29	.14	.040	12.72	3.95	1.12
	1					5062	1186	137	28	.024	.004	.66	.02	.006	1.15	.14	.040	16.78	5.09	1.45
	0					8057	2033	241	56	.031	.006	0.64	.02	.006	0.96	0.13	.037	23.38	6.78	1.93

Fig. 8. Experimental results for synthetic policies with varying $N_{\text{cns}}^{\min}$.

an ACL policy, added noise, and applied our policy mining algorithm to the resulting policy. To add a specified level ν of permission noise, measured as a percentage of $|UP_0|$, we added $\nu|UP_0|/6$ under-assignments and $5\nu|UP_0|/6$ over-assignments to the ACL policy generated from the ABAC policy. This ratio is based on the ratio of Type I and Type II errors in [11, Table 1]. The over-assignments are user-permission tuples generated by selecting the user, resource, and operation from categorical distributions with approximately normally distributed probabilities ("approximately" because the normal distribution is truncated on the sides to have the appropriate finite domain); we adopted this approach from [11]. The under-assignments are removals of user-permission tuples generated in the same way. For each noise level, we ran our policy mining algorithm with noise detection inside a loop that searched for the best values of α (considering values between 0.01 and 0.09 in steps of .01) and τ (considering 0.08, values between 0.1 and 0.9 in steps of 0.1, and between 1 and 10 in steps of 1), because we expect τ to depend on the noise level, and we want to simulate an experienced administrator, so that the results reflect the capabilities and limitations of the noise detection technique rather than the administrator. The best values of α and τ are the ones that maximize the Jaccard similarity of the actual (injected) noise and the reported noise. ROC curves that illustrate the trade-off between false positives and false negatives when tuning the values of α and τ appear in Section 15 in the Supplemental Material.

We started with the university (with $N_{\text{dept}} = 4$), health care (with $N_{\text{ward}} = 6$), and project management (with $N_{\text{dept}} = 6$) sample policies with synthetic attribute data (we also did some experiments with larger policy instances and got similar results), and with synthetic policies with $N_{\text{rule}} = 20$. Figure 10 shows the Jaccard similarity of the actual and reported over-assignments and the Jaccard similarity of the actual and reported under-assignments. Note that, for a policy mining algorithm without noise detection (hence the reported noise is the empty set), these Jaccard similarities would be 0. Each data point is an average over 10 policies, and error bars (too small to see in some cases, and omitted when the standard deviation is 0) show 95% confidence intervals using Student's t-distribution. Over-assignment detection is accurate, with average Jaccard similarity always 0.94 or higher (in our experiments). Under-assignment detection is very good for university and project management, with average Jaccard similarity always 0.93 or higher, but less accurate for health care and synthetic policies, with average Jaccard similarity always 0.63 or higher. Intuitively, detecting over-assignments is somewhat easier, because it is unlikely that there are high-quality rules that cover the over-assignments, so we mostly get rules that do not over-assign and hence the over-assignments get classified correctly. However, under-assignments are more likely to affect the generated rules, leading to mis-classification of under-assignments. As a function of noise level in the considered range, the Jaccard similarities are flat in some cases and generally trend slightly downward in other cases. Figure 11 shows the semantic similarity of the original and mined policies. Note that, for a policy mining algorithm without noise detection, the semantic

N_{rule}	P_{over}	$\|U\|$		$\|R\|$		$\|UP\|$		$\widehat{\|[\![\rho]\!]\|}$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
50	0	693	37	247	15	5246	1445	92.3	30.0	.029	.006	.74	.02	.006	1.16	.11	.03	12.39	6.34	1.80
	0.5	655	53	216	16	7325	1838	333	69.6	.039	.008	.73	.03	.009	1.18	.21	.06	11.64	4.95	1.41
	1	664	58	216	16	7094	1473	563	109	.038	.006	.71	.03	.009	1.23	.29	.08	11.34	4.08	1.16

Fig. 9. Experimental results for synthetic policies with varying P_{over}.

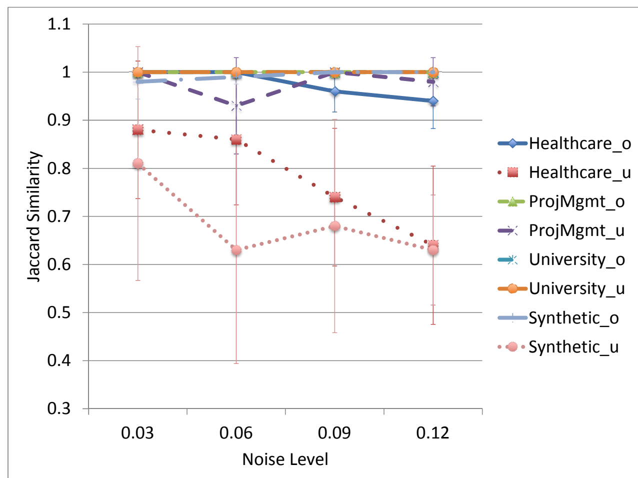

Fig. 10. Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level. Curve names ending with _o and _u are for over-assignments and under-assignments, respectively. The curves for University_u and Synthetic_o are nearly the same and overlap each other.

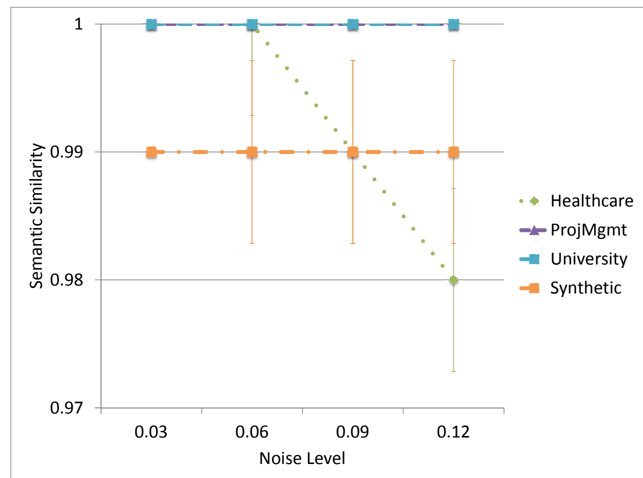

Fig. 11. Semantic similarity of the original policy and the mined policy, as a function of permission noise level.

similarity would equal $1-\nu$. With our algorithm, the semantic similarity is always significantly better than this. The average semantic similarity is always 0.98 or higher, even for $\nu = 0.12$. The similarities are generally lower for synthetic policies than sample policies, as expected, because synthetic policies are not reconstructed as well even in the absence of noise.

Permission Noise and Attribute Noise: To evaluate the effectiveness of our noise detection techniques in the presence of permission noise and attribute noise, we performed experiments in which, for a given noise level ν, we added $\nu|UP_0|/7$ under-assignments, $5\nu|UP_0|/7$ over-assignments, and $\nu|UP_0|/7$ permission errors due to attribute errors to the ACL policy generated from the ABAC policy (in other words, we add attribute errors until $\nu|UP_0|/7$ user-permission tuples have been added or removed due to attribute errors; this way, attribute errors are measured on the same scale as under-assignments and over-assignments). The attribute errors are divided equally between missing values (i.e., replace a non-bottom value with bottom) and incorrect values (i.e., replace a non-bottom value with another non-bottom value). Our current techniques do not attempt to distinguish permission noise from attribute noise (this is a topic for future research); policy analysts are responsible for determining whether a reported suspected error is due to an incorrect permission, an incorrect or missing attribute value, or a false alarm. Since our techniques report only suspected under-assignments and suspected over-assignments, when comparing actual noise to reported noise, permission changes due to attribute noise (i.e., changes in the set of user-permission tuples that satisfy the original policy rules) are included in the actual noise. We started with the same policies as above. Graphs of Jaccard similarity of actual and reported noise, and syntactic similarity of original and mined policies, appear in Section 16 in the Supplemental Material. The results are similar to those without attribute noise, except with slightly lower similarities for the same fraction of permission errors. This shows that our approach to noise detection remains appropriate in the presence of combined attribute noise and permission noise.

5.5 Comparison with Inductive Logic Programming

We implemented a translation from ABAC policy mining to Inductive Logic Programming (ILP) and applied Progol [14], [15], a well-known ILP system developed by Stephen Muggleton, to translations of our sample policies and synthetic policies. Details of the translation appear in Section 17 in the Supplemental Material. Progol mostly succeeds in reconstructing the policies for university and project management, except it fails to learn rules with conjuncts or operation sets containing

multiple constants, instead producing multiple rules. In addition, Progol fails to reconstruct two rules in the health care sample policy. Due to Progol's failure to learn rules with conjuncts or operation sets containing multiple constants, we generated a new set of 20 synthetic policies with at most 1 constant per conjunct and 1 operation per rule. On these policies with $N_{\text{rule}} = 5$, our algorithm achieves a compression factor of 1.92, compared to 1.67 for Progol.

Progol is much slower than our algorithm. For the university (with $N_{\text{dept}} = 10$), health care (with $N_{\text{ward}} = 20$), and project management (with $N_{\text{dept}} = 20$) sample policies, Progol is 302, 375, and 369 times slower than our algorithm, respectively. For synthetic policies with $N_{\text{rule}} = 5$, Progol is 2.74 times slower than our algorithm; for synthetic policies with $N_{\text{rule}} = 10$, we stopped Progol after several hours.

6 Related Work

To the best of our knowledge, the algorithm in this paper is the first policy mining algorithm for any ABAC framework. Existing algorithms for access control policy mining produce role-based policies; this includes algorithms that use attribute data, e.g., [7], [16], [17]. Algorithms for mining meaningful RBAC policies from ACLs and user attribute data [7], [17] attempt to produce RBAC policies that are small (i.e., have low WSC) and contain roles that are meaningful in the sense that the role's user membership is close to the meaning of some user attribute expression. User names (i.e., values of uid) are used in role membership definitions and hence are not used in attribute expressions, so some sets of users cannot be characterized exactly by a user attribute expression. The resulting role-based policies are often much larger than attribute-based policies, due to the lack of parameterization; for example, they require separate roles for each department in an organization, in cases where a single rule suffices in an attribute-based policy. Furthermore, algorithms for mining meaningful roles does not consider resource attributes (or permission attributes), constraints, or set relationships.

Xu and Stoller's work on mining parameterized RBAC (PRBAC) policies [18] is more closely related. Their PRBAC framework supports a simple form of ABAC, because users and permissions have attributes that are implicit parameters of roles, the set of users assigned to a role is specified by an expression over user attributes, and the set of permissions granted to a role is specified by an expression over permission attributes. Our work differs from theirs in both the policy framework and the algorithm. Regarding the policy framework, our ABAC framework supports a richer form of ABAC than their PRBAC framework does. Most importantly, our framework supports multi-valued (also called "set-valued") attributes and allows attributes to be compared using set membership, subset, and equality; their PRBAC framework does not support multi-valued attributes, and it allows attributes to be compared using only equality. Multi-valued attributes are very important in real policies. Due to the lack of multi-valued attributes, the sample policies in [18] contain artificial limitations, e.g., a faculty teaches only one course, and a doctor is a member of only one medical team. Our sample policies are extensions of their case studies without these limitations: a faculty may teach multiple courses, a doctor may be a member of multiple medical teams, etc. Our algorithm works in a different, and more efficient, way than theirs. Our algorithm directly constructs rules to include in the output. Their algorithm constructs a large set of candidate roles and then determines which roles to include in the output, possibly discarding many candidates (more than 90% for their sample policies).

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [10]. Specifically, they use supervised machine learning algorithms to learn classifiers that associate permissions with roles, given as input the permissions, the roles, attribute data for the permissions, and (as training data) the role-permission assignment. The resulting classifier—a support vector machine (SVM)—can be used to automate assignment of new permissions to roles. They also consider a similar scenario in which a supervised machine learning algorithm is used to learn classifiers that associate users with roles, given as input the users, the roles, user attribute data, and the user-role assignment. The resulting classifiers are analogous to attribute expressions, but there are many differences between their work and ours. The largest difference is that their approach needs to be given the roles and the role-permission or user-role assignment as training data; in contrast, our algorithm does 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, but it could be extended to do so. Exploring ABAC policy mining algorithms based on machine learning is a direction for future work.

Lim *et al.* investigated the use of evolutionary algorithms to learn and evolve security policies policies [19]. They consider several problems, including difficult problems related to risk-based policies, but not general ABAC policy mining. In the facet of their work most similar to ABAC policy mining, they showed that genetic programming can learn the access condition in the Bell-LaPadula multi-level security model for mandatory access control. The learned predicate was sometimes syntactically more complex than, but logically equivalent to, the desired predicate.

Association rule mining has been studied extensively. Seminal work includes Agrawal *et al.*'s algorithm for mining propositional rules [20]. Association rule mining algorithms are not well suited to ABAC policy mining, because they are designed to find rules that are probabilistic in nature [20] 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 (to reduce overall policy size).

Bauer *et al.* use association rule mining to detect policy errors [21]. They apply propositional association rule mining to access logs to learn rules expressing that a user who exercised certain permissions is likely to exercise another permission. A suspected misconfiguration exists if a user who exercised the former permissions does not have the latter permission. Bauer *et al.* do not consider attribute data or generate entire policies.

Inductive logic programming (ILP) is a form of machine learning in which concepts are learned from examples and expressed as logic programs. ABAC policies can be represented as logic programs, so ABAC policy mining can be seen as a special case of ILP. However, ILP systems are not ideally suited to ABAC policy mining. ILP is a more difficult problem, which involves learning incompletely specified relations from a limited number of positive and negative examples, exploiting background knowledge, etc. ILP algorithms are correspondingly more complicated and less scalable, and focus more on how much to generalize from the given examples than on optimization of logic program size. For example, Progol (*cf.* Section 5.5) uses a compression (rule size) metric to guide construction of each rule but does not attempt to achieve good compression for the learned rules collectively; in particular, it does not perform steps analogous to merging rules, eliminating overlap between rules, and selecting the highest-quality candidate rules for the final solution. As the experiments in Section 5.5 demonstrate, Progol is slower and generally produces policies with higher WSC, compared to our algorithm.

7 Conclusion

This paper presents an ABAC policy mining algorithm. Experiments with sample policies and synthetic policies demonstrate the algorithm's effectiveness. Directions for future work include supporting additional ABAC policy language features and exploring use of machine learning for ABAC policy mining.

Acknowledgments: We thank the anonymous reviewers for the thorough and detailed reviews that helped us improve this paper.

References

[1] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations (final draft)," National Institute of Standards and Technology, NIST Special Publication 800-162, Sep. 2013.

[2] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[3] NextLabs, Inc., "Managing role explosion with attribute based access control," Jul. 2013, http://www.slideshare.net/nextlabs/managing-role-explosion-with-attribute-based-access-control.

[4] 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.

[5] S. Hachana, N. Cuppens-Boulahia, and F. Cuppens, "Role mining to assist authorization governance: How far have we gone?" *International Journal of Secure Software Engineering*, vol. 3, no. 4, pp. 45–64, October-December 2012.

[6] M. Beckerle and L. A. Martucci, "Formal definitions for usable access control rule sets—From goals to metrics," in *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS)*. ACM, 2013, pp. 2:1–2:11.

[7] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo, "Mining roles with multiple objectives," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 4, 2010.

[8] H. Lu, J. Vaidya, and V. Atluri, "Optimal Boolean matrix decomposition: Application to role engineering," in *Proc. 24th International Conference on Data Engineering (ICDE)*. IEEE, 2008, pp. 297–306.

[9] M. Frank, A. P. Streich, D. A. Basin, and J. M. Buhmann, "A probabilistic approach to hybrid role mining," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 101–111.

[10] Q. Ni, J. Lobo, S. Calo, P. Rohatgi, and E. Bertino, "Automating role-based provisioning by learning from examples," in *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2009, pp. 75–84.

[11] I. Molloy, N. Li, Y. A. Qi, J. Lobo, and L. Dickens, "Mining roles with noisy data," in *Proc. 15th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2010, pp. 45–54.

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

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

[14] S. Muggleton and C. H. Bryant, "Theory completion using inverse entailment," in *Proc. 10th International Conference on Inductive Logic Programming (ILP)*, J. Cussens and A. M. Frisch, Eds. Springer, 2000, pp. 130–146.

[15] S. H. Muggleton and J. Firth, "CProgol4.4: a tutorial introduction," in *Relational Data Mining*, S. Dzeroski and N. Lavrac, Eds. Springer-Verlag, 2001, pp. 160–188.

[16] A. Colantonio, R. Di Pietro, and N. V. Verde, "A business-driven decomposition methodology for role mining," *Computers & Security*, vol. 31, no. 7, pp. 844–855, October 2012.

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

[18] ——, "Mining parameterized role-based policies," in *Proc. Third ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2013.

[19] Y. T. Lim, "Evolving security policies," Ph.D. dissertation, University of York, 2010.

[20] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 487–499.

[21] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," in *Proc. 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2008, pp. 185–194.

Zhongyuan Xu Zhongyuan Xu received the B.S. and M.S. degrees in Computer Science from Peking University in 2006 and 2009, respectively, and the Ph.D. degree in Computer Science from Stony Brook University in 2014. He is currently a software developer at Facebook.

Scott D. Stoller Scott D. Stoller received the B.A. degree in Physics, summa cum laude, from Princeton University in 1990 and the Ph.D. degree in Computer Science from Cornell University in 1997. He is currently a Professor at Stony Brook University.

8 Proof of NP-Hardness

This section shows that the ABAC policy mining problem is NP-hard, by reducing the Edge Role Mining Problem (Edge RMP) [8] to it.

An *RBAC policy* is a tuple $\pi_{\mathrm{RBAC}} = \langle U, P, R, UA, PA\rangle$, where R is a set of roles, $UA \subseteq U \times R$ is the user-role assignment, and $PA \subseteq R \times P$ is role-permission assignment. The *number of edges* in an RBAC policy π_{RBAC} with this form is $|UA|+|PA|$. The user-permission assignment induced by an RBAC policy with the above form is $[\![\pi_{\mathrm{RBAC}}]\!] = UA \circ PA$, where $\circ$ denotes relational composition.

The *Edge Role Mining Problem (Edge RMP)* is [8]: Given an ACL policy $\langle U, P, UP\rangle$, where U is a set of users, P is a set of permissions, and $UP \subseteq U \times P$ is a user-permission relation, find an RBAC policy $\pi_{\mathrm{RBAC}} = \langle U, P, R, UA, PA\rangle$ such that $[\![\pi_{\mathrm{RBAC}}]\!] = UP$ and π_{RBAC} has minimum number of edges among RBAC policies satisfying this condition. NP-hardness of Edge RMP follows from Theorem 1 in [7], since Edge RMP corresponds to the Weighted Structural Complexity Optimization (WSCO) Problem with $w_r = 0$, $w_u = 1$, $w_p = 1$, $w_h = \infty$, and $w_d = \infty$.

Given an Edge RMP problem instance $\langle U, P, UP\rangle$, consider the ABAC policy mining problem instance with ACL policy $\pi_0 = \langle U \cup \{u_0\}, P \cup \{r_0\}, \{op_0\}, UP_0\rangle$, where u_0 is a new user and r_0 is a new resource, $UP_0 = \{\langle u, r, op_0\rangle \mid \langle u, r\rangle \in UP\}$, user attributes $A_{\mathrm{u}} = \{\mathrm{uid}\}$, resource attributes $A_{\mathrm{r}} = \{\mathrm{rid}\}$, user attribute data d_{u} defined by $d_{\mathrm{u}}(u, \mathrm{uid}) = u$, resource attribute data d_{r} defined by $d_{\mathrm{r}}(r, \mathrm{rid}) = r$, and policy quality metric Q_{pol} defined by WSC with $w_1 = 1$, $w_2 = 1$, $w_3 = 0$, and $w_4 = 1$. Without loss of generality, we assume $U \cap P = \emptyset$; this should always hold, because in RBAC, users are identified by names that are atomic values, and permissions are resource-operation pairs; if for some reason this assumption doesn't hold, we can safely rename users or permissions to satisfy this assumption, because RBAC semantics is insensitive to equalities between users and permissions.

A solution to the given Edge-RMP problem instance can be constructed trivially from a solution π_{ABAC} to the above ABAC policy mining instance by interpreting each rule as a role. Note that rules in π_{ABAC} do not contain any constraints, because uid and rid are the only attributes, and $U \cap P = \emptyset$ ensures that constraints relating uid and rid are useless (consequently, any non-zero value for w_4 suffices). The presence of the "dummy" user u_0 and "dummy" resource r_0 ensure that the UAE and RAE in every rule in π_{ABAC} contains a conjunct for uid or rid, respectively, because no correct rule can apply to all users or all resources. These observations, and the above choice of weights, implies that the WSC of a rule ρ in π_{RBAC} equals the number of users that satisfy ρ plus the number of resources (i.e., permissions) that satisfy ρ. Thus, $\mathrm{WSC}(\pi_{\mathrm{RBAC}})$ equals the number of edges in the corresponding RBAC policy, and an ABAC policy with minimum WSC corresponds to an RBAC policy with minimum number of edges.

9 Asymptotic Running Time

This section analyzes the asymptotic running time of our algorithm. We first analyze the main loop in Figure 1, i.e., the while loop in lines 3–11. First consider the cost of one iteration. The running time of candidateConstraint in line 5 is $O(|A_{\mathrm{u}}| \times |A_{\mathrm{r}}|)$. The running time of line 6 is $O(|U_{r,o}| \times |A_{\mathrm{u}}| \times |A_{\mathrm{r}}|)$, where $U_{r,o} = \{u' \in U \mid \langle u', r, o\rangle \in uncovUP\}$; this running time is achieved by incrementally maintaining an auxiliary map that maps each pair $\langle r, o\rangle$ in $R \times Op$ to $U_{r,o}$. The running time of function generalizeRule in line 4 in Figure 2 is $O(|2^{|cc|}|)$. Other steps in the main loop are either constant time or linear, i.e., $O(|A_{\mathrm{u}}| + |A_{\mathrm{r}}| + |UP_0|)$. Now consider the number of iterations of the main loop. The number of iterations is $|Rules_1|$, where $Rules_1$ is the set of rules generated by the main loop. In the worst case, the rule generated in each iteration covers one user-permission tuple, and $|Rules_1|$ is as large as $|UP_0|$. Typically, rules generalize to cover many user-permission tuples, and $|Rules_1|$ is much smaller than $|UP_0|$.

The running time of function mergeRules is $O(|Rules_1|^3)$. The running time of function simplifyRules is based on the running times of the five "elim" functions that it calls. Let $lc_{\mathrm{u,m}}$ (mnemonic for "largest conjunct") denote the maximum number of sets in a conjunct for a multi-valued user attribute in the rules in $Rules_1$, i.e., $\forall a \in A_{\mathrm{u,m}}.\, \forall \rho \in Rules_1.\, |\mathrm{uae}(\rho)(a)| \leq lc_{\mathrm{u,m}}$. The value of $lc_{\mathrm{u,m}}$ is at most $|\mathrm{Val_m}|$ but typically small (one or a few). The running time of function elimRedundantSets is $O(|A_{\mathrm{u}}| \times lc_{\mathrm{u,m}}^2 \times |\mathrm{Val_s}|)$. Checking validity of a rule ρ takes time linear in $|[\![\rho]\!]|$. Let lm (mnemonic for "largest meaning") denote the maximum value of $|[\![\rho]\!]|$ among all rules ρ passed as the first argument in a call to elimConstraints, elimConjuncts, or elimElements. The value of lm is at most $|UP_0|$ but typically much smaller. The running time of function elimConstraints is $O((2^{|cc|}) \times lm)$. The running time of function elimConjuncts is $O((2^{|A_{\mathrm{u}}|} + 2^{|A_{\mathrm{r}}|}) \times lm)$. The exponential factors in the running time of elimConstraints and elimConjuncts are small in practice, as discussed above; note that the factor of lm represents the cost of checking validity of a rule. The running time of elimElements is $O(|A_{\mathrm{u}}| \times lm)$. Let le (mnemonic for "largest expressions") denote the maximum of $\mathrm{WSC}(\mathrm{uae}(\rho)) + \mathrm{WSC}(\mathrm{rae}(\rho))$ among rules ρ contained in any set $Rules$ passed as the first argument in a call to simplifyRules. The running time of elimOverlapVal is $O(|Rules_1| \times (|A_{\mathrm{u}}| + |A_{\mathrm{r}}|) \times le)$. The running time of elimOverlapOp is $O(|Rules_1| \times |Op| \times le)$. The factor le in the running times of elimOverlapVal and elimOverlapOp represents the cost of subset checking. The number of iterations of the while loop in line 13–15 is $|Rules_1|$ in the worst case. The overall running time of the algorithm is worst-case cubic in $|UP_0|$.

10 Processing Order

This section describes the order in which tuples and rules are processed by our algorithm.

When selecting an element of $uncovUP$ in line 4 of the top-level pseudocode in Figure 1, the algorithm selects the user-permission tuple with the highest (according to lexicographic order) value for the following quality metric Q_{up}, which maps user-permission tuples to triples. Informally, the first two components of $Q_{\mathrm{up}}(\langle u, r, o\rangle)$ are the frequency of permission p and user u, respectively, i.e., their numbers of occurrences in UP_0, and the third component is the string representation of $\langle u, r, o\rangle$ (a deterministic although somewhat arbitrary tie-breaker when the first two components of the metric are equal).

$$\begin{aligned}
\mathrm{freq}(\langle r, o\rangle) &= |\{\langle u', r', o'\rangle \in UP_0 \mid r' = r \wedge o' = o\}| \\
\mathrm{freq}(u) &= |\{\langle u', r', o'\rangle \in UP_0 \mid u' = u\}| \\
Q_{\mathrm{up}}(\langle u, r, o\rangle) &= \langle \mathrm{freq}(\langle r, o\rangle), \mathrm{freq}(u), \mathrm{toString}(\langle \mathrm{u}, \mathrm{r}, \mathrm{o}\rangle)\rangle
\end{aligned}$$

In the iterations over $Rules$ in mergeRules and simplifyRules, the order in which rules are processed is deterministic in our implementation, because $Rules$ is implemented as a linked list, loops iterate over the rules in the order they appear in the list, and newly generated rules are added at the beginning of the list. In mergeRules, the workset is a priority queue sorted in descending lexicographic order of rule pair quality, where the quality of a rule pair $\langle \rho_1, \rho_2\rangle$ is $\langle \max(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2)), \min(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2))\rangle$.

11 Optimizations

Periodic Merging of Rules.: Our algorithm processes UP_0 in batches of 1000 tuples, and calls mergeRules after processing each batch. Specifically, 1000 tuples are selected at random from $uncovUP$, they are processed in the order described in Section 10, mergeRules($Rules$) is called, and then another batch of tuples is processed.

This heuristic optimization is motivated by the observation that merging sometimes has the side-effect of generalization, i.e., the merged rule may cover more tuples than the rules being merged. Merging earlier (compared to waiting until $uncovUP$ is empty) allows additional tuples covered by merged rules to be removed from $uncovUP$ before those tuples are processed by the loop over $uncovUP$ in the top-level pseudocode in Figure 1. Without this heuristic optimization, those tuples would be processed by the loop over $uncovUP$, additional rules would be generated from them, and those rules would probably later get merged with other rules, leading to the same policy.

Caching: To compute $[\![\rho]\!]$ for a rule ρ, our algorithm first computes $[\![\mathrm{uae}(\rho)]\!]$ and $[\![\mathrm{rae}(\rho)]\!]$. As an optimization, our implementation caches $[\![\rho]\!]$, $[\![\mathrm{uae}(\rho)]\!]$, and $[\![\mathrm{rae}(\rho)]\!]$ for each rule ρ. Each of these values is stored after the first time it is computed. Subsequently, when one of these values is needed, it is recomputed only if some component of ρ, $\mathrm{uae}(\rho)$ or $\mathrm{rae}(\rho)$, respectively, has changed. In our experiments, this optimization improves the running time by a factor of approximately 8 to 10.

Early Stopping.: In the algorithm without noise detection, in mergeRules, when checking validity of ρ_{merge}, our algorithm does not compute $[\![\rho_{\mathrm{merge}}]\!]$ completely and then test $[\![\rho_{\mathrm{merge}}]\!] \subseteq UP_0$. Instead, as it computes $[\![\rho_{\mathrm{merge}}]\!]$, it immediately checks whether each element is in UP_0, and if not, it does not bother to compute the rest of $[\![\rho_{\mathrm{merge}}]\!]$. In the algorithm with noise detection, that validity test is replaced with the test $|\,[\![\rho_{\mathrm{merge}}]\!] \setminus UP_0| \div |\,[\![\rho_{\mathrm{merge}}]\!]\,| \leq \alpha$. We incrementally compute the ratio on the left while computing $[\![\rho_{\mathrm{merge}}]\!]$, and if the ratio exceeds 2α, we stop computing $[\![\rho_{\mathrm{merge}}]\!]$, under the assumption that ρ_{merge} would probably fail the test if we continued. This heuristic decreases the running time significantly. It can affect the result, but it had no effect on the result for the problem instances on which we evaluated it.

12 Details of Sample Policies

The figures in this section contain all rules and some illustrative attribute data for each sample policy. This section also describes in more detail the manually written attribute datasets and synthetic attribute datasets for the sample policies.

The policies are written in a concrete syntax with the following kinds of statements. userAttrib($uid, a_1 = v_1, a_2 = v_2, \ldots$) provides user attribute data for a user whose "uid" attribute equals *uid* and whose attributes $a_1, a_2, \ldots$ equal $v_1, v_2, \ldots$, respectively. The resourceAttrib statement is similar. The statement rule(*uae*; *pae*; *ops*; *con*) defines a rule; the four components of this statement correspond directly to the four components of a rule as defined in Section 2. In the attribute expressions and constraints, conjuncts are separated by commas. In constraints, the superset relation "$\supseteq$" is denoted by ">", the contains relation "$\ni$" is denoted by "]", and the superset-of-an-element-of relation $\supseteq\in$ is denoted by "supseteqIn".

University Sample Policy: In our university sample policy, user attributes include position (applicant, student, faculty, or staff), department (the user's department), crsTaken (set of courses taken by a student), crsTaught (set of courses for which the user is the instructor (if the user is a faculty) or the TA (if the user is a student), and isChair (true if the user is the chair of his/her department). Resource attributes include type (application, gradebook, roster, or transcript), crs (the course a gradebook or roster is for, for those resource types), student (the student whose transcript or application this is, for type=transcript or type=application), and department (the department the course is in, for type $\in$ {gradebook, roster}; the student's major department, for type=transcript). The policy rules and illustrative userAttrib and resourceAttrib statements appear in Figure 12. The constraint "crsTaken] crs" in the first rule for gradebooks ensures that a user can apply the readMyScores operation only to gradebooks for courses

the student has taken. This is not essential, but it is natural and is advisable according to the defense-in-depth principle.

The manually written attribute dataset for this sample policy contains a few instances of each type of user and resource: two academic departments, a few faculty, a few gradebooks, several students, a few staff in each of two administrative departments (admissions office and registrar), etc. We generated a series of synthetic attribute datasets, parameterized by the number of academic departments. The generated userAttrib and resourceAttrib statements for the users and resources associated with each department are similar to but more numerous than the userAttrib and resourceAttrib statements in the manually written dataset. For each department, a scaling factor called the *department size* is selected from a normal distribution with mean 1 and standard deviation 1, truncated at 0.5 and 5 (allowing a 10-to-1 ratio between the sizes of the largest and smallest departments). The numbers of applicants, students, faculty, and courses associated with a department equal the department size times n_{app}, n_{stu}, n_{fac}, and n_{crs}, respectively, where $n_{\mathrm{app}} = 5$, $n_{\mathrm{stu}} = 20$, $n_{\mathrm{fac}} = 5$, $n_{\mathrm{crs}} = 10$. The numbers of courses taught by faculty, taken by students, and TAd by students are selected from categorical distributions with approximately normally distributed probabilities. The courses taught by faulty are selected uniformly. The courses taken and TAd by students are selected following a Zipf distributions, to reflect the varying popularity of courses. The number of staff in each administrative department is proportional to the number of academic departments.

Health Care Sample Policy: In our health care sample policy, user attributes include position (doctor or nurse; for other users, this attribute equals $\perp$), specialties (the medical areas that a doctor specializes in), teams (the medical teams a doctor is a member of), ward (the ward a nurse works in or a patient is being treated in), and agentFor (the patients for which a user is an agent). Resource attributes include type (HR for a health record, or HRitem for a health record item), patient (the patient that the HR or HR item is for), treatingTeam (the medical team treating the aforementioned patient), ward (the ward in which the aforementioned patient is being treated), author (author of the HR item, for type=HRitem), and topics (medical areas to which the HR item is relevant, for type=HRitem). The policy rules and illustrative userAttrib and resourceAttrib statements appear in Figure 13.

The manually written attribute dataset for this sample policy contains a small number of instances of each type of user and resource: a few nurses, doctors, patients, and agents, two wards, and a few items in each patient's health record. We generated a series of synthetic attribute datasets, parameterized by the number of wards. The generated userAttrib and resourceAttrib statements for the users and resources associated with each ward are similar to but more numerous than the userAttrib and resourceAttrib statements in the manually written dataset. For each ward, a scaling factor called the *ward size* is selected from a normal distribution with the same parameters as the department size distribution

```
// Rules for Gradebooks
// A user can read his/her own scores in gradebooks
// for courses he/she has taken.
rule(; type=gradebook; readMyScores; crsTaken ] crs)
// A user (the instructor or TA) can add scores and
// read scores in the gradebook for courses he/she
// is teaching.
rule(; type=gradebook; {addScore, readScore};
    crsTaught ] crs;)
// The instructor for a course (i.e., a faculty teaching
// the course) can change scores and assign grades in
// the gradebook for that course.
rule(position=faculty; type=gradebook;
    {changeScore, assignGrade}; crsTaught ] crs)

// Rules for Rosters
// A user in registrar's office can read and modify all
// rosters.
rule(department=registrar; type=roster; {read, write}; )
// The instructor for a course (i.e., a faculty teaching
// the course) can read the course roster.
rule(position=faculty; type=roster; {read};
    crsTaught ] crs)

// Rules for Transcripts
// A user can read his/her own transcript.
rule(; type=transcript; {read}; uid=student)
// The chair of a department can read the transcripts
// of all students in that department.
rule(isChair=true; type=transcript; {read};
    department=department)
// A user in the registrar's office can read every
// student's transcript.
rule(department=registrar; type=transcript; {read}; )

// Rules for Applications for Admission
// A user can check the status of his/her own application.
rule(; type=application; {checkStatus}; uid=student)

// A user in the admissions office can read, and
// update the status of, every application.
rule(department=admissions; type=application;
    {read, setStatus}; )

// An illustrative user attribute statement.
userAttrib(csFac2, position=faculty, department=cs,
        crsTaught={cs601})
// An illustrative resource attribute statement.
resourceAttrib(cs601gradebook, department=cs,
            crs=cs601, type=gradebook)
```

Fig. 12. University sample policy.

described above. The numbers of patients, nurses, doctors, agents, and teams associated with a ward equal the ward size times n_{pat}, n_{nurse}, n_{doc}, n_{ag}, and n_{team}, respectively, where $n_{\text{pat}} = 10$, $n_{\text{doc}} = 2$, $n_{\text{nurse}} = 4$, $n_{\text{ag}} = 2$, and $n_{\text{team}} = 2$. The numbers of items in each patient's medical record, the topics associated with each HR item, patients associated with each agent, specialties per doctor, and teams each doctor is on are selected from categorical distributions with approximately normally distributed probabilities. The topics and specialties associated with HR items and doctors, respectively, are selected following a Zipf distribution.

Project Management Sample Policy: In our project management sample policy, user attributes include projects (projects the user is working on), projectsLed (projects led by the user), adminRoles (the user's administrative roles, e.g., accountant, auditor, planner, manager), expertise (the user's areas of technical expertise, e.g., design, coding), tasks (tasks assigned to the user), department (department that the user is in), and isEmployee (true if the user is an employee, false if the user is a contractor). Resource attributes include type (task, schedule, or budget), project (project that the task, schedule, or budget is for), department (department that the aforementioned project is in), expertise (areas of technical expertise required to work on the task, for type=task) and proprietary (true if the task involves proprietary information, which is accessible only to employees, not contractors). The policy rules and illustrative userAttrib and resourceAttrib statements appear in Figure 14.

The manually written attribute dataset for this sample policy contains a small number of instances of each type of user (managers, accountants, coders, non-employees (e.g,, contractors) and employees with various areas of expertise, etc.) and each type of resource (two departments, two projects per department, three tasks per project, etc.). We generated a series of synthetic attribute datasets, parameterized by the number of departments. The generated userAttrib and resourceAttrib statements for the users and resources associated with each department are similar to but more numerous than the userAttrib and resourceAttrib statements in the manually written dataset. For each department, a scaling factor called the *department size* is selected from a normal distribution with the same parameters as for the university sample policy. The numbers of projects, non-employees per area of expertise, and employees per area of expertise that are associated with a department equal the department size times n_{proj}, n_{nonEmp}, and n_{emp}, respectively, where $n_{\text{proj}} = 2$, $n_{\text{nonEmp}} = 1$, and $n_{\text{emp}} = 1$. Each department has one manager, accountant, auditor, and planner, and one project leader per project. Six tasks are associated with each project. The projects assigned to non-employees and employees are selected following a Zipf distribution.

```
// Rules for Health Records
// A nurse can add an item in a HR for a patient in
// the ward in which he/she works.
rule(position=nurse; type=HR; {addItem}; ward=ward)
// A user can add an item in a HR for a patient treated
// by one of the teams of which he/she is a member.
rule(; type=HR; {addItem}; teams ] treatingTeam)
// A user can add an item with topic "note" in his/her
// own HR.
rule(; type=HR; {addNote}; uid=patient)
// A user can add an item with topic "note" in the HR
// of a patient for which he/she is an agent.
rule(; type=HR; {addNote}; agentFor ] patient)

// Rules for Health Record Items
// The author of an item can read it.
rule(; type=HRitem; {read}; uid=author)
// A nurse can read an item with topic "nursing" in a HR
// for a patient in the ward in which he/she works.
rule(position=nurse; type=HRitem,
     topics supseteqIn {{nursing}}; {read}; ward=ward)
// A user can read an item in a HR for a patient treated
// by one of the teams of which he/she is a member, if
// the topics of the item are among his/her specialties.
rule(; type=HRitem; {read}; specialties > topics,
     teams ] treatingTeam)
// A user can read an item with topic "note" in his/her
// own HR.
rule(; type=HRitem, topics supseteqIn {{note}}; {read};
     uid=patient)
// An agent can read an item with topic "note" in the
// HR of a patient for which he/she is an agent.
rule(; type=HRitem, topics supseteqIn {{note}}; {read};
     agentFor ] patient)
// An illustrative user attribute statement.
userAttrib(oncDoc1, position=doctor,
           specialties={oncology},
           teams={oncTeam1, oncTeam2})
// An illustrative resource attribute statement.
resourceAttrib(oncPat1nursingItem, type=HRitem,
               author=oncNurse2, patient=oncPat1,
               topics={nursing}, ward=oncWard,
               treatingTeam=oncTeam1)
```

Fig. 13. Health care sample policy.

13 Example: Processing of a User-Permission Tuple

Figure 15 illustrates the processing of the user-permission tuple $t = \langle \text{csFac2}, \text{addScore}, \text{cs601gradebook} \rangle$ selected as a seed (i.e., selected in line 4 of Figure 1), in a smaller version of the university sample policy containing only one rule, namely, the second rule in Figure 12. Attribute data for user csFac2 and resource cs601gradebook appear in Figure 12.

The edge from t to cc labeled "$\text{candidateConstraint}$" represents the call to $\text{candidateConstraint}$, which re-

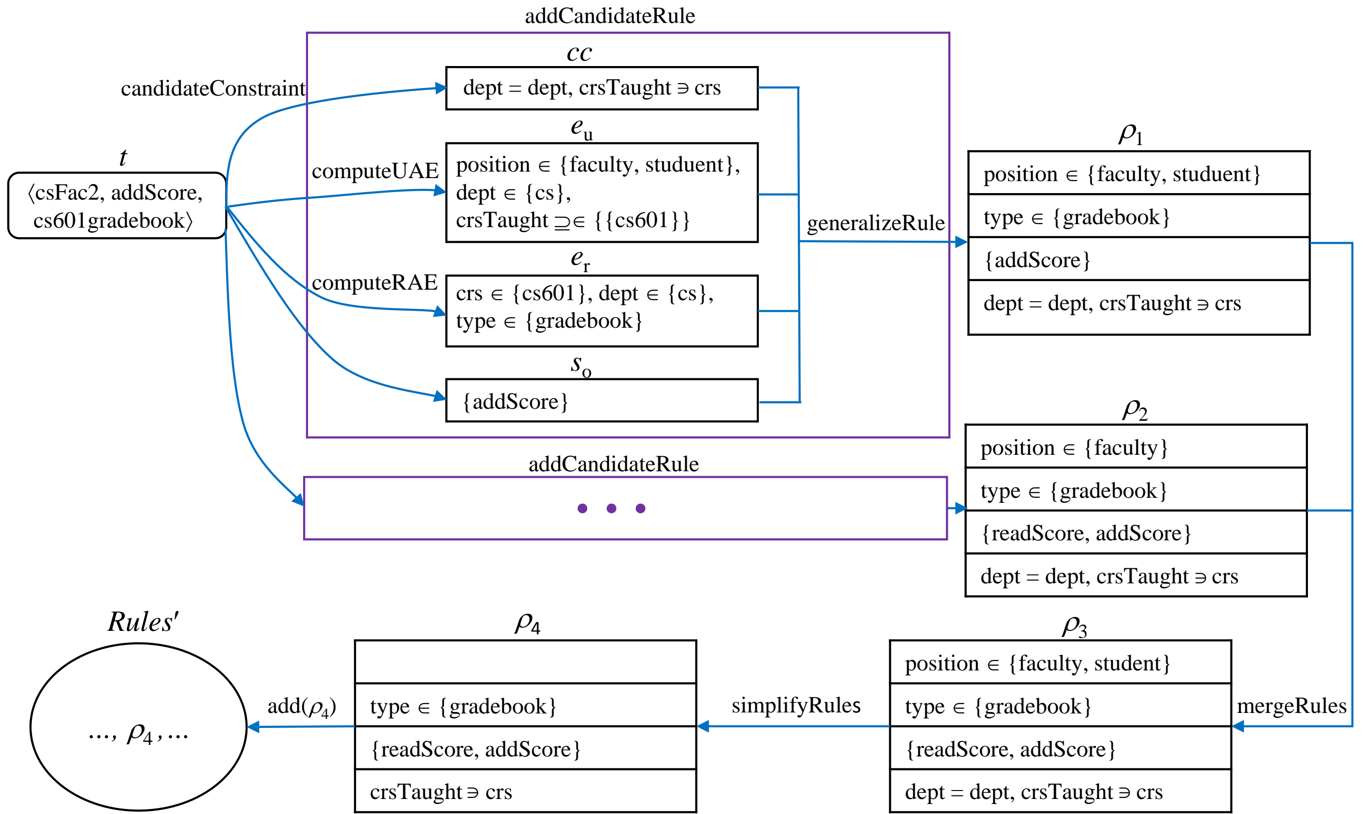

Fig. 15. Diagram representing the processing of one user-permission tuple selected as a seed, in the university sample policy. Rules are depicted as rectangles with four compartments, corresponding to the four components of a rule tuple.

turns the set of atomic constraints that hold between csFac2 and cs601gradebook; these constraints are shown in the box labeled cc. The two boxes labeled "$\mathrm{addCandidateRule}$" represent the two calls to $\mathrm{addCandidateRule}$. Internal details are shown for the first call but elided for the second call. The edges from t to e_{u} and from t to e_{r} represent the calls in $\mathrm{addCandidateRule}$ to $\mathrm{computeUAE}$ and $\mathrm{computeRAE}$, respectively. The call to $\mathrm{computeUAE}$ returns a user-attribute expression e_{u} that characterizes the set s_{u} containing users u' with permission $\langle \mathrm{addScore}, \mathrm{cs601gradebook}\rangle$ and such that $\mathrm{candidateConstraint}(\mathrm{cs601gradebook}, u') = cc$. The call to $\mathrm{computeRAE}$ returns a resource-attribute expression that characterizes $\{\mathrm{cs601gradebook}\}$. The set of operations considered in this call to $\mathrm{addCandidateRule}$ is simply $s_{\mathrm{o}} = \{\mathrm{addScore}\}$. The call to $\mathrm{generalizeRule}$ generates a candidate rule ρ_1 by assigning e_{u}, e_{r} and s_{o} to the first three components of ρ_1, and adding the two atomic constraints in cc to ρ_1 and eliminating the conjuncts in e_{u} and e_{r} corresponding to the attributes mentioned in cc. Similarly, the second call to $\mathrm{addCandidateRule}$ generates another candidate rule ρ_2. The call to $\mathrm{mergeRules}$ merges ρ_1 and ρ_2 to form ρ_3, which is simplified by the call to $\mathrm{simplifyRules}$ to produce a simplified rule ρ_4, which is added to candidate rule set Rules'.

14 Syntactic Similarity

Syntactic similarity of policies measures the syntactic similarity of rules in the policies. The *syntactic similarity* of rules ρ and ρ', denoted $\mathrm{ss}(\rho, \rho')$, is defined by

$$\begin{aligned}
\mathrm{ss}_{\mathrm{u}}(e, e') &= |A_{\mathrm{u}}|^{-1} \sum_{a \in A_{\mathrm{u}}} J(e(a), e'(a)) \\
\mathrm{ss}_{\mathrm{r}}(e, e') &= |A_{\mathrm{r}}|^{-1} \sum_{a \in A_{\mathrm{r}}} J(e(a), e'(a)) \\
\mathrm{ss}(\rho, \rho') &= \mathrm{mean}(\mathrm{ss}_{\mathrm{u}}(\mathrm{uae}(\rho), \mathrm{uae}(\rho')), \mathrm{ss}_{\mathrm{r}}(\mathrm{rae}(\rho), \mathrm{rae}(\rho')), \\
&\qquad J(\mathrm{ops}(\rho), \mathrm{ops}(\rho')),\ J(\mathrm{con}(\rho), \mathrm{con}(\rho')))
\end{aligned}$$

where the Jaccard similarity of two sets is $J(S_1, S_2) = |S_1 \cap S_2| / |S_1 \cup S_2|$.

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 is the maximum of the syntactic similarity of the sets of rules in the policies, considered in both orders (this makes the relation symmetric).

$$\begin{aligned}
\mathrm{ss}(\mathit{Rules}, \mathit{Rules}') &= |\mathit{Rules}|^{-1} \times \\
&\quad \textstyle\sum_{\rho \in \mathit{Rules}} \max(\{\mathrm{ss}(\rho, \rho') \mid \rho' \in \mathit{Rules}'\}) \\
\mathrm{ss}(\pi, \pi') &= \max(\mathrm{ss}(\mathrm{rules}(\pi), \mathrm{rules}(\pi')), \\
&\qquad \mathrm{ss}(\mathrm{rules}(\pi'), \mathrm{rules}(\pi)))
\end{aligned}$$

```
// The manager of a department can read and approve
// the budget for a project in the department.
rule(adminRoles supseteqIn {{manager}}; type=budget;
    {read approve}; department=department)
// A project leader can read and write the project
// schedule and budget.
rule( ; type in {schedule, budget}; {read, write};
    projectsLed ] project)
// A user working on a project can read the project
// schedule.
rule( ; type=schedule; {read}; projects ] project)
// A user can update the status of tasks assigned to
// him/her.
rule( ; type=task; {setStatus}; tasks ] rid)
// 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.
rule( ; type=task, proprietary=false; {read request};
    projects ] project, expertise > expertise)
// An employee working on a project can read and
// request to work on any task whose required areas
// of expertise are among his/her areas of expertise.
rule(isEmployee=True; type=task; {read request};
    projects ] project, expertise > expertise)
// An auditor assigned to a project can read the
// budget.
rule(adminRoles supseteqIn {{auditor}}; type=budget;
    {read}; projects ] project)
// An accountant assigned to a project can read and
// write the budget.
rule(adminRoles supseteqIn {{accountant}};
    type=budget; {read, write}; projects ] project)
// An accountant assigned to a project can update the
// cost of tasks.
rule(adminRoles supseteqIn {{accountant}}; type=task;
    {setCost}; projects ] project)
// A planner assigned to a project can update the
// schedule.
rule(adminRoles supseteqIn {{planner}};
    type=schedule; {write}; projects ] project)
// A planner assigned to a project can update the
// schedule (e.g., start date, end date) of tasks.
rule(adminRoles supseteqIn {{planner}}; type=task;
    {setSchedule}; projects ] project)
// An illustrative user attribute statement.
userAttrib(des11, expertise={design}, projects={proj11},
    isEmployee=True,
    tasks={proj11task1a, proj11task1propa})
// An illustrative resource attribute statement.
resourceAttrib(proj11task1a, type=task, project=proj11,
    department=dept1, expertise={design},
    proprietary=false)
```

Fig. 14. Project management sample policy.

15 ROC Curves for Noise Detection Parameters

When tuning the parameters α and τ used in noise detection (see Section 4.1), there is a trade-off between true positives and false positives. To illustrate the trade-off, the Receiver Operating Characteristic (ROC) curve in Figure 16 shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on α and τ for synthetic policies with 20 rules and 6% noise, split between under-assignments and over-assignments as described in Section 5.4. Figure 17 shows the TPR and FPR for over-assignments. Each data point is an average over 10 synthetic policies. In each of these two sets of experiments, true positives are reported noise (of the specified type, i.e., over-assignments or under-assignments) or that are also actual noise; false negatives are actual noise that are not reported; false positives are reported noise that are not actual noise; and true negatives are user-permission tuples that are not actual noise and are not reported as noise.

Generally, we can see from the ROC curves that, with appropriate parameter values, it is possible to achieve very high TPR and FPR simultaneously, so there is not a significant inherent trade-off between them.

From the ROC curve for under-assignments, we see that the value of τ does not affect computation of under-assignments, as expected, because detection of under-assignments is performed before detection of over-assignments (the former is done when each rule is generated, and the latter is done at the end). We see from the diagonal portion of the curve in the upper left that, when choosing the value of α, there is a trade-off between the TPR and FPR, i.e., having a few false negatives and a few false positives.

From the ROC curve for over-assignments, we see that the value of α affects the rules that are generated, and hence it affects the computation of over-assignments based on those rules at the end of the rule generation process. For $\alpha = 0.01$, when choosing τ, there is some trade-off between the TPR and FPR. For $\alpha \geq 0.02$, the FPR equals 0 independent of τ, so there is no trade-off: the best values of τ are the ones with the highest TPR.

16 Graphs of Results from Experiments with Permission Noise and Attribute Noise

For the experiments with permission noise and attribute noise described in Section 5.4, Figure 18 shows the Jaccard similarity of the actual and reported over-assignments and the Jaccard similarity of the actual and reported under-assignments, and Figure 19 shows the semantic similarity of the original and mined policies. Each data point is an average over 10 policies. Error bars show 95% confidence intervals using Student's t-distribution.

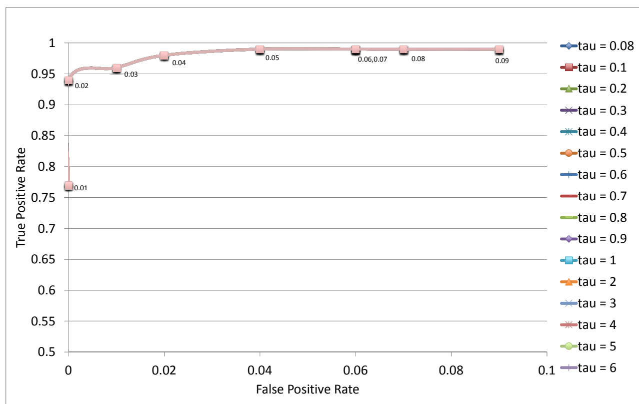

Fig. 16. ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on α and τ.

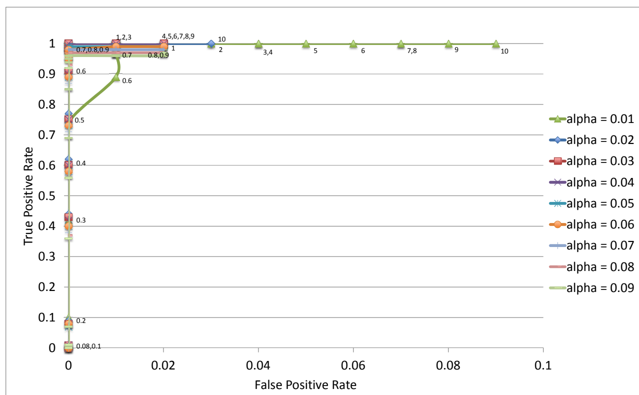

Fig. 17. ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for over-assignments on α and τ.

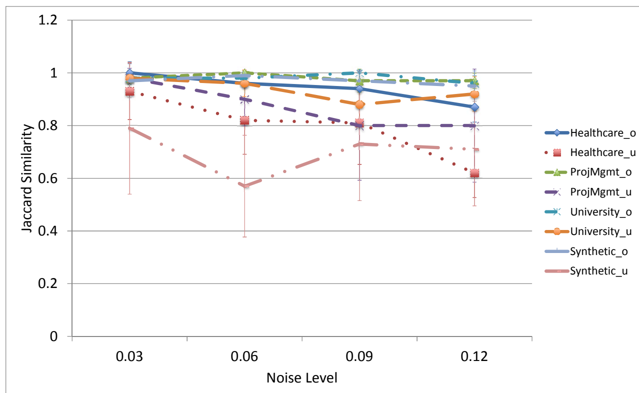

Fig. 18. Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level due to permission noise and attribute noise.

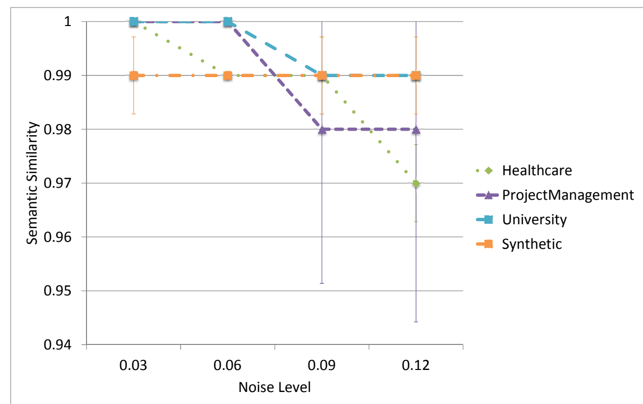

Fig. 19. Semantic similarity of the original policy and the mined policy, as a function of permission noise level due to permission noise and attribute noise.

17 TRANSLATION TO INDUCTIVE LOGIC PROGRAMMING

This section describes our translation from the ABAC policy mining problem to inductive logic programming (ILP) as embodied in Progol [14], [15]. Given an ACL policy and attribute data, we generate a Progol input file, which contains type definitions, mode declarations, background knowledge, and examples.

Type Declarations: Type definitions define categories of objects. The types `user`, `resource`, `operation`, and `attribValAtomic` (corresponding to $\mathrm{Val_s}$) are defined by a statement for each constant of that type; for example, for each user u, we generate the statement `user(`u`)`. The type `attribValSet` (corresponding to $\mathrm{Val_m}$) is defined by the rules

```
attribValSet([]).
attribValSet([V|Vs]) :- attribValAtomic(V),
                        attribValSet(Vs).
```

For each attribute a, we define a type containing the constants that appear in values of that attribute in the attribute data; for example, for each value d of the "department" attribute, we generate the statement `departmentType(`d`)`.

Mode Declarations: Mode declarations restrict the form of rules that Progol considers, by limiting how each predicate may be used in learned rules. Each head mode declaration `modeh(...)` describes a way in which a predicate may be used in the head (conclusion) of a learned rule. Each body mode declaration `modeb(...)` describes a way in which a predicate may be used in the body (premises) of a learned rule. Each mode declaration has two arguments. The second argument specifies, for each argument a of the predicate, the type of a and whether a may be instantiated with an input variable (indicated by "+"), an output variable (indicated by "−"), or a constant (indicated by "#"). The first argument, called the *recall*, is an integer or $\star$, which bounds the number of values of the output arguments for which the predicate can hold

for given values of the input arguments and constant arguments; "$\star$" indicates no bound. The specification of predicate arguments as inputs and outputs also limits how variables may appear in learned rules. In a learned rule h :- $b_1, \ldots, b_n$, every variable of input type in each premise b_i must appear either with input type in h or with output type in some premise b_j with $j < i$.

We generate only one head mode declaration:

```
modeh(1, up(+user, +resource, #operation))
```

This tells Progol to learn rules that define the user-permission predicate up.

For each single-valued user attribute a, we generate a body mode declaration modeb(1, aU(+user, #aType)). For example, the mode declaration for a user attribute named "department" is modeb(1, departmentU(+user, #departmentType)). We append "U" to the attribute name to prevent naming conflicts in case there is a resource attribute with the same name. Mode declarations for multi-valued user attributes are defined similarly, except with "$\star$" instead of 1 as the recall. Mode declarations for resource attributes are defined similarly, except with R instead of U appended to the attribute name. We tried a variant translation in which we generated a second body mode declaration for each attribute, using -aType instead of #aType, but this led to worse results.

We also generate mode declarations for predicates used to express constraints. For each single-valued user attribute a and single-valued resource attribute $\bar{a}$, we generate a mode declaration modeb(1, aU_equals_$\bar{a}$R(+user,+resource)); the predicate aU_equals_$\bar{a}$R is used to express atomic constraints of the form $a = \bar{a}$. The mode declarations for the predicates used to express the other two forms of atomic constraints are similar, using user and resource attributes with appropriate cardinality, and with "contains" (for $\ni$) or "superset" (for $\supseteq$) instead of "equals" in the name of the predicate.

Background Knowledge: The attribute data is expressed as background knowledge. For each user u and each single-valued user attribute a, we generate a statement aU(u,v) where $v = d_{\mathrm{u}}(u, a)$. For each user u and each multi-valued user attribute a, we generate a statement aU(u,v) for each $v \in d_{\mathrm{u}}(u, a)$. Background knowledge statements for resource attribute data are defined similarly.

Definitions of the predicates used to express constraints are also included in the background knowledge. For each equality predicate a_equals_$\bar{a}$ mentioned in the mode declarations, we generate a statement aU_equals_$\bar{a}$R(U,R) :- aU(U,X), $\bar{a}$R(R,X). The definitions of the predicates used to express the other two forms of constraints are

```
aU_contains_āR(U,R) :- aU(U,X), āR(R,X).
aU_superset_āR(U,R) :- setof(X,aU(U,X),SU),
                        setof(Y,āR(R,Y),SR),
                        superset(SU,SR),
                        not(SR==[]).
superset(Y,[A|X]) :- element(A,Y),
                     superset(Y,X).
superset(Y,[]).
```

The premise not(SR==[]) in the definition of aU_superset_$\bar{a}$R is needed to handle cases where the value of $\bar{a}$ is $\perp$. The predicates setof and element are built-in predicates in Progol.

Examples: A *positive example* is an instantiation of a predicate to be learned for which the predicate holds. A *negative example* is an instantiation of a predicate to be learned for which the predicate does not hold. For each $\langle u, r, o\rangle \in U \times R \times Op$, if $\langle u, r, o\rangle \in UP_0$, then we generate a positive example up(u,r,o), otherwise we generate a negative example :- up(u,r,o) (the leading ":-" indicates that the example is negative). The negative examples are necessary because, without them, Progol may produce rules that hold for instantiations of up not mentioned in the positive examples.