# 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 ACL policy or 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.

## 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 role-based access control (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 case studies and on synthetic 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

Several ABAC frameworks have been proposed, varying in administrative model, flexibility, and expressiveness. A brief survey appears in [7]. 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 for 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 $A_{\mathrm{u}}$ of user attributes, user attribute data is represented by a function

*This material is based upon work supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grant CNS-0831298. Submitted to IEEE TDSC.* 

$d_\mathrm{u}$ such that $d_\mathrm{u}(u,a)$ is the value of attribute $a$ for user $u$. There is a distinguished user attribute $\mathrm{uid}$ that has a unique value for each user. Similarly, given a set $R$ of resources and a set $A_\mathrm{r}$ of resource attributes, resource attribute data is represented by a function $d_\mathrm{r}$ such that $d_\mathrm{r}(r,a)$ is the value of attribute $a$ for resource $r$. There is a distinguished resource attribute $\mathrm{rid}$ that has a unique value for each resource. We assume the set $A_\mathrm{u}$ of user attributes can be partitioned into a set $A_{\mathrm{u},1}$ of *single-valued user attributes* which have atomic values, and a set $A_{\mathrm{u,m}}$ of *multi-valued user attributes* whose values are sets of atomic values. Similarly, we assume the set $A_\mathrm{r}$ of resource attributes can be partitioned into a set $A_{\mathrm{r},1}$ of *single-valued resource attributes* and a set of $A_{\mathrm{r,m}}$ of *multi-valued resource attributes*. Let $\mathrm{Val_s}$ be the set of possible atomic values of attributes. We assume $\mathrm{Val_s}$ 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 $\mathrm{Val_m} = \mathrm{Set}(\mathrm{Val_s} \setminus \{\bot\}) \cup \bot$, where $\mathrm{Set}(S)$ 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 $\mathrm{Val_s} \setminus \{\bot\}$ or $\mathrm{Val_m} \setminus \{\bot\}$, 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 $\mathrm{attr}(e)$ denote the set of attributes used by $e$. Let $\mathrm{attr}_1(e)$ and $\mathrm{attr_m}(e)$ 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 $(\forall a \in A_{\mathrm{u},1}.\, e(a) = \top \vee \exists v \in e(a).\, d_\mathrm{u}(u,a) = v)$ and $(\forall a \in A_{\mathrm{u,m}}.\, e(a) = \top \vee \exists v \in e(a).\, d_\mathrm{u}(u,a) \supseteq v)$. For multi-valued attributes, we use the condition $d_\mathrm{u}(u,a) \supseteq v$ instead of $d_\mathrm{u}(u,a) = v$ because elements of a multi-valued user attribute typically represent some type of capabilities of a user, so using $\supseteq$ expresses that the user has the specified capabilities and possibly more.

For example, suppose $A_{\mathrm{u},1} = \{\mathrm{dept}, \mathrm{position}\}$ and $A_{\mathrm{u,m}} = \{\mathrm{courses}\}$. The function $e_1$ with $e_1(\mathrm{dept}) = \{\mathrm{CS}\}$ and $e_1(\mathrm{position}) = \{\mathrm{grad}, \mathrm{ugrad}\}$ and $e_1(\mathrm{courses}) = \{\{\mathrm{CS101}, \mathrm{CS102}\}\}$ is a user-attribute expression satisfied by users in the CS department who are either graduate or undergraduate students and whose courses include CS101 and CS102 (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 $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{ugrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq \{\mathrm{CS101}, \mathrm{CS102}\}$. For an example that uses $\supseteq\!\in$, the expression $e_2$ that is the same as $e_1$ except with $e_2(\mathrm{courses}) = \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$ may be written as $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{ugrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq\!\in \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$, and is satisfied by graduate or undergraduate students in the CS department whose courses include either CS101 or CS102.

The *meaning* of a user-attribute expression $e$, denoted $[\![e]\!]_U$, is the set of users in $U$ that satisfy it: $[\![e]\!]_U = \{u \in U \mid u \models e\}$. 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 $A_\mathrm{r}$ of resource attributes instead of the set $A_\mathrm{u}$ 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 $a_{\mathrm{u,m}} \supseteq a_{\mathrm{r,m}}$, $a_{\mathrm{u,m}} \ni a_{\mathrm{r},1}$, or $a_{\mathrm{u},1} = a_{\mathrm{r},1}$, where $a_{\mathrm{u},1} \in A_{\mathrm{u},1}$, $a_{\mathrm{u,m}} \in A_{\mathrm{u,m}}$, $a_{\mathrm{r},1} \in A_{\mathrm{r},1}$, and $a_{\mathrm{r,m}} \in A_{\mathrm{r,m}}$. The first two forms express that user attributes contain specified values. This is a common type of constraint, because user attributes typically represent some type of capabilities of a user. Other forms of atomic constraint are possible (e.g., $a_{\mathrm{u,m}} \subseteq a_{\mathrm{r,m}}$) but less common, so we leave them for future work. Let $\mathrm{uAttr}(f)$ and $\mathrm{rAttr}(f)$ refer to the user attribute and resource attribute, respectively, used in $f$. User $u$ and resource $r$ *satisfy* an atomic constraint $f$, denoted $\langle u, r\rangle \models f$, if $d_\mathrm{u}(u, \mathrm{uAttr}(f)) \neq \bot$ and $d_\mathrm{r}(u, \mathrm{rAttr}(f)) \neq \bot$ and formula $f$ holds when the values $d_\mathrm{u}(u, \mathrm{uAttr}(f))$ and $d_\mathrm{r}(u, \mathrm{rAttr}(f))$ are substituted in it.

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

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

A *rule* is a tuple $\langle e_\mathrm{u}, e_\mathrm{r}, O, c\rangle$, where $e_\mathrm{u}$ is a user-attribute expression, $e_\mathrm{r}$ is a resource-attribute expression, $O$ is a set of operations, and $c$ is a constraint. For a rule $\rho = \langle e_\mathrm{u}, e_\mathrm{r}, O, c\rangle$, let $\mathrm{uae}(\rho) = e_\mathrm{u}$, $\mathrm{rae}(\rho) = e_\mathrm{r}$, $\mathrm{ops}(\rho) = O$, and $\mathrm{con}(\rho) = c$. For example, the rule $\langle$true, type=task $\wedge$ proprietary=false, {read, request}, projects

$\ni$ project $\wedge$ expertise $\supseteq$ expertise$\rangle$ used in our project management case study can be interpreted as "A user working on a project can read and request to work on a non-proprietary task whose required areas of expertise are among his/her areas of expertise." User $u$, resource $r$, and operation $o$ *satisfy* a rule $\rho$, denoted $\langle u, r, o\rangle \models \rho$, if $u \models \mathrm{uae}(\rho) \wedge r \models \mathrm{rae}(\rho) \wedge o \in \mathrm{ops}(\rho) \wedge \langle u, r\rangle \models \mathrm{con}(\rho)$.

An *ABAC policy* is a tuple $\langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, Rules\rangle$, where $U$, $R$, $A_{\mathrm{u}}$, $A_{\mathrm{r}}$, $d_{\mathrm{u}}$, and $d_{\mathrm{r}}$ 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 $\rho$ is $[\![\rho]\!] = \{\langle u, r, o\rangle \in U \times R \times Op \mid \langle u, r, o\rangle \models \rho\}$. Note that $U$, $R$, $d_{\mathrm{u}}$, and $d_{\mathrm{r}}$ are implicit arguments to $[\![\rho]\!]$.

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

## 3 The ABAC Policy Mining Problem

An *access control list (ACL) policy* is a tuple $\langle U, R, Op, UP_0\rangle$, where $U$ is a set of users, $R$ is a set of resources, $Op$ is a set of operations, and $UP_0 \subseteq U \times R \times Op$ is a user-permission relation, obtained from the union of the access control lists.

An ABAC policy $\pi$ is *consistent* with an ACL policy $\langle U, P, Op, UP_0\rangle$ if they have the same sets of users, resource, and operations and $[\![\pi]\!] = UP_0$.

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 $\pi_0$, 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 $\pi_0$ contains rules that use them.

The other criterion is to maximize a policy quality metric. A *policy quality metric* is a function $Q_{\mathrm{pol}}$ 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 $\pi_0 = \langle U, R, Op, UP_0\rangle$, user attributes $A_{\mathrm{u}}$, resource attributes $A_{\mathrm{r}}$, user attribute data $d_{\mathrm{u}}$, resource attribute data $d_{\mathrm{r}}$, and a policy quality metric $Q_{\mathrm{pol}}$, find a set $Rules$ of rules such that the ABAC policy $\pi = \langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, Rules\rangle$ that (1) is consistent with $\pi_0$, (2) uses uid only when necessary, (3) uses rid only when necessary, and (4) has the best quality, according to $Q_{\mathrm{pol}}$, among such policies.

The policy quality metric that our algorithm aims to optimize is *weighted structural complexity* (WSC) [8], 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 $\pi$ with rules $Rules$ is $\mathrm{WSC}(\pi) = \mathrm{WSC}(Rules)$, defined by

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

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) [9] to it. NP-hardness of Edge RMP follows from Theorem 1 in [8]. 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 $\pi_{\mathrm{ABAC}}$ 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

The high-level design of our policy mining algorithm is sketched in Section 1. For efficiency, our algorithm incorporates heuristics and is not guaranteed to generate a policy with minimal WSC. Top-level pseudo-code for our algorithm appears in Figure 1. We refer to tuples selected in line 4 as *seeds*. The top-level pseudo-code calls several functions, described next. Function names hyperlink to pseudocode for the function, if it is included in the paper, otherwise to the description of the function. An example illustrating the processing of a user-permission tuple by our algorithm appears in Section 11 in the Supplemental Material.

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

The function $\mathrm{computeUAE}(s, U)$ in Figure 3 computes a user-attribute expression $e_{\mathrm{u}}$ that characterizes the set $s$ of users. Preference is given to attribute expressions that do not use uid, since attribute-based policies are generally preferable to identity-based policies, even when they have higher WSC, because attribute-based generalize better. After constructing a candidate expression $e$, it calls $\mathrm{elimRedundantSets}(e)$, which attempts to lower the WSC of $e$ by examining the conjunct for each multi-valued user attribute, and removing each set that is a superset of another set in the same conjunct; this leaves the meaning of the rule unchanged, because $\supseteq$ is used in the condition for multi-valued attributes in the semantics of user attribute expressions. Pseudocode for elimRedundantSets is straightforward and omitted. The expression $e_{\mathrm{u}}$ returned by computeUAE might not be minimum-sized among expressions that characterize $s$: it is possible that some attributes mapped to a set of values by $e_{\mathrm{u}}$ can instead be mapped to $\top$. We defer minimization of $e_{\mathrm{u}}$ until after the call to generalizeRule (described below), because minimizing $e_{\mathrm{u}}$ before that would reduce opportunities to find relations between values of user attributes and resource attributes in generalizeRule.

The function computeRAE is defined in the same way as computeUAE, except using resource attributes instead of user attributes, and the call to elimRedundantSets is omitted. Pseudocode for computeRAE is omitted.

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

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

The function $\mathrm{generalizeRule}(\rho, cc, uncovUP, Rules)$ in Figure 4 attempts to generalize rule $\rho$ by adding some of the atomic constraints $f$ in $cc$ to $\rho$ 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 $\rho$. Such a rule is more general than $\rho$ in the sense that it refers to relationships instead of specific values. Also, the user-permission relation induced by a generalization of $\rho$ is a superset of the user-permission relation induced by $\rho$.

If there are no valid generalizations of $\rho$, then $\mathrm{generalizeRule}(\rho, cc, uncovUP, Rules)$ simply returns $\rho$. If there is a valid generalization of $\rho$, $\mathrm{generalizeRule}(\rho, cc, uncovUP, Rules)$ returns the generalization $\rho'$ of $\rho$ with the best quality according to a given rule quality metric. Note that $\rho'$ 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 $Q_{\mathrm{rul}}(\rho, UP)$ that maps a rule $\rho$ 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

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

In generalizeRule, $uncovUP$ is the second argument to $Q_{\mathrm{rul}}$, so $[\![\rho]\!] \cap UP$ is the set of user-permission tuples in $UP_0$ that are covered by $\rho$ 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 $\rho$. The function calls itself recursively to determine the subsequent atomic constraints in $c$ that get added to the constraint.

*// Rules is the set of candidate rules*
$Rules = \emptyset$
*// uncovUP contains user-permission tuples in* $UP_0$
*// that are not covered by Rules*
$uncovUP = UP_0.\mathrm{copy}()$
**while** $\neg uncovUP.\mathrm{isEmpty}()$
  *// Select an uncovered user-permission tuple.*
  $\langle u, r, o \rangle$ = some tuple in $uncovUP$
  $cc = \mathrm{candidateConstraint}(r, u)$
  *//* $s_{\mathrm{u}}$ *contains users with permission* $\langle r, o \rangle$ *and*
  *// that have the same candidate constraint for* $r$ *as* $u$
  $s_{\mathrm{u}} = \{u' \in U \mid \langle u', r, o \rangle \in UP_0$
      $\wedge\ \mathrm{candidateConstraint}(r, u') = cc\}$
  $\mathrm{addCandidateRule}(s_{\mathrm{u}}, \{r\}, \{o\}, cc, uncovUP, Rules)$
  *//* $s_{\mathrm{o}}$ *is set of operations that* $u$ *can apply to* $r$
  $s_{\mathrm{o}} = \{o' \in Op \mid \langle u, r, o' \rangle \in UP_0\}$
  $\mathrm{addCandidateRule}(\{u\}, \{r\}, s_{\mathrm{o}}, cc, uncovUP, Rules)$
**end while**
*// Repeatedly merge and simplify rules, until*
*// this has no effect*
$\mathrm{mergeRules}(Rules)$
**while** $\mathrm{simplifyRules}(Rules)$ && $\mathrm{mergeRules}(Rules)$
  skip
**end while**
*// Select high quality rules into final result* $Rules'$.
$Rules' = \emptyset$
Repeatedly select the highest quality rules
from $Rules$ to $Rules'$, according to $Qrul$,
until $\sum_{\rho \in Rules'} [\![\rho]\!] = UP_0$
**return** $Rules'$

Fig. 1. Policy mining algorithm.

### 4.1 Functions to Merge and Simplify Rules

The function $\mathrm{mergeRules}(Rules)$ in Figure 5 attempts to reduce the WSC of $Rules$ by removing redundant rules and merging pairs of rules. A rule $\rho$ in $Rules$ is *redundant* if $Rules$ contains another rule $\rho'$ such that

**function** addCandidateRule($s_{\mathrm{u}}, s_r, s_{\mathrm{o}}, cc, \mathit{uncovUP}, \mathit{Rules}$)
// *Construct a rule* $\rho$ *that covers user-permission*
// *tuples* $\{\langle u, r, o\rangle \mid u \in s_{\mathrm{u}} \wedge r \in s_r \wedge o \in s_{\mathrm{o}}\}$.
$e_{\mathrm{u}} = \mathrm{computeUAE}(s_{\mathrm{u}}, U)$
$e_{\mathrm{r}} = \mathrm{computeRAE}(s_r, R)$
$\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, s_{\mathrm{o}}, \emptyset\rangle$
$\rho' = \mathrm{generalizeRule}(\rho, cc, \mathit{uncovUP}, \mathit{Rules})$
$\mathit{Rules}.\mathrm{add}(\rho')$
$\mathit{uncovUP}.\mathrm{removeAll}([\![\rho']\!])$

Fig. 2. Compute a candidate rule $\rho'$ and add $\rho'$ to candidate rule set $\mathit{Rules}$

**function** computeUAE($s, U$)
// *Try to characterize* $s$ *without using* uid. *Use all*
// *other attributes which have known values for all*
// *users in* $s$.
$e = (\lambda\, a \in A_{\mathrm{u}}.$
    $a = \mathrm{uid} \vee (\exists\, u \in s.\, d_{\mathrm{u}}(u, a) = \bot)\, ?\, \top : \bigcup_{u \in s} d_{\mathrm{u}}(u, a))$
**if** $[\![e]\!]_U \neq s$
    // uid *is needed to characterize* $s$
    $e(\mathrm{uid}) = \bigcup_{u \in s} d_{\mathrm{u}}(u, \mathrm{uid})$
**end if**
$\mathrm{elimRedundantSets}(e)$
**return** $e$

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

$[\![\rho]\!] \subseteq [\![\rho']\!]$. Informally, rules $\rho_1$ and $\rho_2$ are merged by taking, for each attribute, the union of the conjuncts in $\rho_1$ and $\rho_2$ for that attribute. If the resulting rule $\rho_{\mathrm{merge}}$ is valid, $\rho_{\mathrm{merge}}$ is added to $\mathit{Rules}$, and $\rho_1$ and $\rho_2$ and any other rules that are now redundant are removed from $\mathit{Rules}$. As an optimization (in the implementation, not reflected in the pseudocode), we do not compute $[\![\rho_{\mathrm{merge}}]\!]$ completely and then test $[\![\rho_{\mathrm{merge}}]\!] \subseteq \mathit{UP}_0$; instead, as we compute $[\![\rho_{\mathrm{merge}}]\!]$, we immediately check whether each element is in $\mathit{UP}_0$, and if not, we do not bother to compute the rest of $[\![\rho_{\mathrm{merge}}]\!]$. mergeRules($\mathit{Rules}$) updates its argument $\mathit{Rules}$ in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules($\mathit{Rules}$) in Figure 6 attempts to simplify all of the rules in $\mathit{Rules}$. It updates its argument $\mathit{Rules}$ in place, replacing rules in $\mathit{Rules}$ with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several ways, which are embodied in the following functions that it calls. The names of these functions start with "elim", because they attempt to eliminate unnecessary parts of rules. To enable simplifyRules to determine whether any rules were simplified, each "elim" function returns a Boolean value indicating whether it simplified any rules. Computation of the Boolean return values of "elim" functions are not reflected in the pseudocode for brevity.

The function elimRedundantSets is described above.

**function** generalizeRule($\rho, cc, \mathit{uncovUP}, \mathit{Rules}$)
// $\rho_{\mathrm{best}}$ *is highest-quality generalization of* $\rho$
$\rho_{\mathrm{best}} = \rho$
// $cc'$ *contains formulas from* $cc$ *that lead to valid*
// *generalizations of* $\rho$.
$cc' =$ new Vector()
// $gen[i]$ *is a generalization of* $\rho$ *using* $cc'[i]$
$gen =$ new Vector()
// *find formulas in* $cc$ *that lead to valid*
// *generalizations of* $\rho$.
**for** $f$ **in** $cc$
    // *try to generalize* $\rho$ *by adding* $f$ *and elimi-*
    // *nating conjuncts for both attributes used in* $f$.
    $\rho' = \langle \mathrm{uae}(\rho)[\mathrm{uAttr}(f) \mapsto \top], \mathrm{rae}(\rho)[\mathrm{rAttr}(f) \mapsto \top],$
        $\mathrm{ops}(\rho), \mathrm{con}(\rho) \cup \{f\}\rangle$
    // *check if* $[\![\rho']\!]$ *is a valid rule*
    **if** $[\![\rho']\!] \subseteq \mathit{UP}_0$
        $cc'.add(f)$
        $gen.add(\rho')$
    **else**
        // *try to generalize* $\rho$ *by adding* $f$ *and elimi-*
        // *nating conjunct for one user attribute used in* $f$
        $\rho' = \langle \mathrm{uae}(\rho)[\mathrm{uAttr}(f) \mapsto \top], \mathrm{rae}(\rho),$
            $\mathrm{ops}(\rho), \mathrm{con}(\rho) \cup \{f\}\rangle$
        **if** $[\![\rho']\!] \subseteq \mathit{UP}_0$
            $cc'.add(f)$
            $gen.add(\rho')$
        **else**
    // *try to generalize* $\rho$ *by adding* $f$ *and elimi-*
    // *nating conjunct for one resource attribute used in* $f$.
            $\rho' = \langle \mathrm{uae}(\rho), \mathrm{rae}(\rho)[\mathrm{rAttr}(f) \mapsto \top],$
                $\mathrm{ops}(\rho), \mathrm{con}(\rho) \cup \{f\}\rangle$
            **if** $[\![\rho']\!] \subseteq \mathit{UP}_0$
                $cc'.add(f)$
                $gen.add(\rho')$
            **end if**
        **end if**
    **end if**
**end for**
**for** $i$ = 1 **to** $cc'$.length
    // *try to further generalize* $gen[i]$
    $\rho'' = \mathrm{generalizeRule}(gen[i], cc'[i{+}1\,..], \mathit{uncovUP},$
        $\mathit{Rules})$
    **if** $Q_{\mathrm{rul}}(\rho'', \mathit{uncovUP}) > Q_{\mathrm{rul}}(\rho_{\mathrm{best}}, \mathit{uncovUP})$
        $\rho_{\mathrm{best}} = \rho''$
    **end if**
**end for**
**return** $\rho_{\mathrm{best}}$

Fig. 4. Generalize rule $\rho$ 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*)
// *Remove redundant rules*
$rdtRules = \{\rho \in Rules \mid \exists\, \rho' \in Rules \setminus \{\rho\}.\ [\![\rho]\!] \subseteq [\![\rho']\!]\}$
$Rules$.removeAll($rdtRules$)
// *Merge rules*
$workSet = \{(\rho_1, \rho_2) \mid \rho_1 \in Rules \wedge \rho_2 \in Rules \wedge \rho_1 \neq \rho_2 \wedge \mathrm{con}(\rho_1) = \mathrm{con}(\rho_2)\}$
**while** not($workSet$.empty())
  // *Remove an arbitrary element of the workset*
  $(\rho_1, \rho_2) = workSet$.remove()
  $\rho_{\mathrm{merge}} = \langle \mathrm{uae}(\rho_1) \cup \mathrm{uae}(\rho_2), \mathrm{rae}(\rho_1) \cup \mathrm{rae}(\rho_2), \mathrm{ops}(\rho_1) \cup \mathrm{ops}(\rho_2), \mathrm{con}(\rho_1)\rangle$
  **if** $[\![\rho_{\mathrm{merge}}]\!] \subseteq UP_0$
    // *The merged rule is valid. Add it to Rules,*
    // *and remove rules that became redundant.*
    $rdtRules = \{\rho \in Rules \mid [\![\rho]\!] \subseteq [\![\rho_{\mathrm{merge}}]\!]\}$
    $Rules$.removeAll($rdtRules$)
    $workSet$.removeAll($\{(\rho_1, \rho_2) \in workSet \mid \rho_1 \in rdtRules \vee \rho_2 \in rdtRules\}$)
    $workSet$.addAll($\{(\rho_{\mathrm{merge}}, \rho) \mid \rho \in Rules \wedge \mathrm{con}(\rho) = \mathrm{con}(\rho_{\mathrm{merge}})\}$)
    $Rules$.add($\rho_{\mathrm{merge}}$)
  **end if**
**end while**
**return** true if any rules were merged

Fig. 5. 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 $e(a) = e_1(a) \cup e_2(a)$.

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

**function** elimConjuncts($\rho$, $Rules$, $UP$)
1: $Au = \{$"user"$\} \times \mathrm{attr}(\mathrm{uae}(\rho)) \setminus A_{\mathrm{unrm}}$
2: $Ar = \{$"res"$\} \times \mathrm{attr}(\mathrm{rae}(\rho)) \setminus A_{\mathrm{unrm}}$
3: **if** maxConjunctSz(uae($\rho$)) $\geq$ maxConjunctSz(rae($\rho$))
4:   $\rho'$ = elimConjunctsHelper($\rho$, $Au$, $UP$)
5:   $\rho''$ = elimConjunctsHelper($\rho'$, $Ar$, $UP$)
6: **else**
7:   $\rho'$ = elimConjunctsHelper($\rho$, $Ar$, $UP$)
8:   $\rho''$ = elimConjunctsHelper($\rho'$, $Au$, $UP$)
9: **end if**
10: **if** $\rho'' \neq \rho$
11:   replace $\rho$ with $\rho''$ in $Rules$
12: **end if**

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

Fig. 6. Functions used to simplify rules.

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

The function elimConjuncts($\rho$, $Rules$, $UP$) in Figure 6 attempts to increase the quality of rule $\rho$ by eliminating some conjuncts. It calls the function elimConjunctsHelper($\rho$, $A$, $UP$) in Figure 6, which considers all rules that differ from $\rho$ by mapping a subset $A'$ of the tagged attributes in $A$ to $\top$ instead of to a set of values; among the resulting rules that are valid, it returns one with the highest quality. A *tagged attribute* is a pair of the form $\langle$"user"$, a\rangle$ with $a \in A_{\mathrm{u}}$ or $\langle$"res"$, a\rangle$ with $a \in A_{\mathrm{r}}$. The set $A_{\mathrm{unrm}}$ in lines 1–2 of elimConjuncts is a set of *unremovable* tagged attributes; it is a parameter of the algorithm, specifying attributes that should not be eliminated, because eliminating them increases the risk of generating an overly general policy, i.e., a policy that might grant inappropriate permissions when new users or new resources (hence new permissions) are added to the system. We use a combinatorial algorithm for elimConjuncts that evaluates all combinations of conjuncts that can be eliminated, because elimination of one conjunct might prevent elimination of another conjunct. This algorithm makes elimConjuncts worst-case exponential in the numbers of user attributes and resource attributes that can be eliminated while preserving validity of the rule; in practice the number of such attributes is small. elimConjuncts also considers whether to remove conjuncts from the user attribute expression or the resource attribute expression first, because elimination of a conjunct in one attribute expression might prevent elimination of a conjunct in the other. The algorithm could simply try both orders, but instead it uses a heuristic that, in our experiments, is faster and almost as effective: if $\mathrm{maxConjunctSz}(e_{\mathrm{u}}) \geq \mathrm{maxConjunctSz}(e_{\mathrm{r}})$ then eliminate conjuncts from the user attribute expression first, otherwise eliminate conjuncts from the resource attribute expression first, where $\mathrm{maxConjunctSz}(e)$ is the size (WSC) of the largest conjunct in attribute expression $e$. elimConjunctsHelper calls the function $\mathrm{elimAttribute}(\rho, ta)$, which returns a copy of rule $\rho$ with the conjunct for tagged attribute $ta$ removed from the user attribute expression or resource attribute expression as appropriate (in other words, the specified attribute is mapped to $\bot$); pseudo-code for elimAttribute is straightforward and omitted.

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

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

The function $\mathrm{elimOverlapVal}(\rho, \mathit{Rules})$ attempts to decrease the WSC of rule $\rho$ by removing values from conjuncts of attribute expressions in $\rho$ if there are other rules that cover the affected user-permission tuples. Specifically, a value $v$ in the conjunct for a user attribute $a$ in $\rho$ is removed if there is another rule $\rho'$ in $\mathit{Rules}$ such that (1) $\mathrm{attr}(\mathrm{uae}(\rho')) \subseteq \mathrm{attr}(\mathrm{uae}(\rho))$ and $\mathrm{attr}(\mathrm{rae}(\rho')) \subseteq \mathrm{attr}(\mathrm{rae}(\rho))$, (2) the conjunct of $\mathrm{uae}(\rho')$ for $a$ contains $v$, (3) each conjunct of $\mathrm{uae}(\rho')$ or $\mathrm{rae}(\rho')$ other than the conjunct for $a$ is either $\top$ or a superset of the corresponding conjunct of $\rho$, and (4) $\mathrm{con}(\rho') \subseteq \mathrm{con}(\rho)$. 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, $\rho$ is removed from $\mathit{Rules}$. $\mathrm{elimOverlapVal}(\rho, \mathit{Rules})$ returns true if it modifies or removes $\rho$, otherwise it returns false. Pseudo-code for elimOverlapVal is straightforward and omitted.

The function $\mathrm{elimOverlapOp}(\rho, \mathit{Rules})$ attempts to decrease the WSC of rule $\rho$ by removing operations from $\mathrm{ops}(\rho)$, if there are other rules that cover the affected user-permission tuples. Specifically, an operation $o$ is removed from $\mathrm{ops}(\rho)$ if there is another rule $\rho'$ in $\mathit{Rules}$ such that (1) $\mathrm{attr}(\mathrm{uae}(\rho')) \subseteq \mathrm{attr}(\mathrm{uae}(\rho))$ and $\mathrm{attr}(\mathrm{rae}(\rho')) \subseteq \mathrm{attr}(\mathrm{rae}(\rho))$, (2) $\mathrm{ops}(\rho')$ contains $o$, (3) each conjunct of $\mathrm{uae}(\rho')$ or $\mathrm{rae}(\rho')$ is either $\top$ or a superset of the corresponding conjunct of $\rho$, and (4) $\mathrm{con}(\rho')$ is a subset of $\mathrm{con}(\rho)$. If $\mathrm{ops}(\rho)$ becomes empty, $\rho$ is removed from $\mathit{Rules}$. $\mathrm{elimOverlapOp}(\rho, \mathit{Rules})$ returns true if it modifies or removes $\rho$, otherwise it returns false. Pseudocode for elimOverlapOp is straightforward and omitted.

### 4.2 Additional Details of the Algorithm

**Iteration Order:** When selecting an element of $\mathit{uncovUP}$ in line 4 of the top-level pseudo-code in Figure 1, the algorithm selects the user-permission tuple with the highest (according to lexicographic order) value for the following quality metric $Q_{\mathrm{up}}$, 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 $\mathit{UP}_0$, 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).

$$\begin{aligned}
\mathrm{freq}(\langle r, o\rangle) &= |\{\langle u', r', o'\rangle \in \mathit{UP}_0 \mid r' = r \wedge o' = o\}| \\
\mathrm{freq}(u) &= |\{\langle u', r', o'\rangle \in \mathit{UP}_0 \mid u' = u\}| \\
Q_{\mathrm{up}}(\langle u, r, o\rangle) &= \langle \mathrm{freq}(\langle r, o\rangle), \mathrm{freq}(u), \mathrm{toString}(\langle \mathrm{u}, \mathrm{r}, \mathrm{o}\rangle)\rangle
\end{aligned}$$

In the iterations over $\mathit{Rules}$ in mergeRules and simplifyRules, the order in which rules are processed is deterministic in our implementation, because $\mathit{Rules}$ 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 $\langle \max(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2)), \min(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2))\rangle$.

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

**Subtle Aspects:** A few aspects of the detailed design of our algorithm are subtle. For example, the initial version of our algorithm constructed only one rule from each selected user-permission tuple, corresponding to the first call to addCandidateRule in Figure 1; we later realized that, for good results, another rule, corresponding to the second call to addCandidateRule in Figure 1, should also be constructed. As another example, the initial version of function generalizeRule in Figure 4 only contained the case containing the first assignment to $\rho'$, in which both conjuncts related to atomic constraint $f$ are eliminated from the attribute expressions; we later realized that it was sometimes useful to generalize the rule by eliminating only one of those conjuncts, so we added the cases containing the other two assignments to $\rho'$.

**Asymptotic Running Time:** The overall running time of the algorithm is worst-case cubic in $|UP_0|$. A detailed analysis of the worst-case asymptotic running time appears in Section 9 in the Supplemental Material. Briefly, in the worst case, the algorithm generates one rule per user-permission tuple, and mergeRules is worst-case cubic in the number of rules. In the experiments with case studies 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 [10], [11]. Therefore, before applying our algorithm to a dataset that might contain irrelevant attributes, it is advisable to use the method in [10] or [12] to determine the relevance of each attribute to the user-permission assignment and then eliminate attributes with low relevance.

### 4.3 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 [12]. 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 [12]. 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 $\tau$. The rule quality metric used here is the first component of the metric used in the loop in Figure 1 that constructs $Rules'$; thus, $\tau$ is a threshold on the value of $Q_{\rm rul}(\rho, uncovUP)$, and the rules with quality less than or equal to $\tau$ 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 $\tau$, and then it removes rules with quality less than or equal to $\tau$ from $Rules'$. Adjustment of $\tau$ 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 $\tau$ until the number of reported over-assignments is that percentage of $|UP_0|$. Note that re-computing over-assignments after a change to $\tau$ does not require re-generating the policy.

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 $\alpha$ quantifies the notion of "relatively small". A rule is $\alpha$ *almost valid* if the fraction of invalid user-permission tuples in $[\![\rho]\!]$ is at most $\alpha$, i.e., $|\,[\![\rho]\!] \setminus UP_0| \leq \alpha \times |\,[\![\rho]\!]\,|$. In places where the policy mining algorithm checks whether a rule is valid, if the rule is $\alpha$ 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 $\alpha$ is guided by the user, similarly as for the over-assignment threshold $\tau$.

## 5 Evaluation

The general methodology used for evaluation is described in Section 1. We applied this methodology to case studies and synthetic policies. We implemented our policy mining algorithm in Java and ran the experiments on a laptop with a 2.5 GHz Intel Core i5 CPU. In our experiments, all of the weights $w_i$ in the definition of WSC equal 1.

### 5.1 Evaluation on Case Studies

We developed four case studies for use in evaluation of our algorithm. The number of rules in the case studies is relatively small, but they express non-trivial policies and exercise all the features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The manually written attribute dataset for each case study contains a small number of instances of each type of user and resource. The case studies are described very briefly in this section. Figure 7 provides information about their size. Details of the case studies, including all policy rules and some illustrative attribute data, appear in Section 10 in the Supplemental Material.

| Case Study | $\|Rules\|$ | $\|U\|$ | $\|R\|$ | $\|Op\|$ | $\|UP\|$ | $\|A_u\|$ | $\|A_r\|$ | $\|\widehat{[\![\rho]\!]}\|$ |
|---|---|---|---|---|---|---|---|---|
| university | 10 | 22 | 34 | 9 | 168 | 6 | 5 | 19.0 |
| healthcare | 9 | 21 | 16 | 3 | 51 | 6 | 7 | 6.7 |
| project mgmt | 11 | 19 | 40 | 7 | 189 | 8 | 6 | 19.0 |
| online video | 6 | 12 | 13 | 1 | 78 | 3 | 3 | 20.0 |

Fig. 7. Sizes of the case studies, with manually written attribute data. $\widehat{|[\![\rho]\!]|}$ is the average number of user-permission tuples that satisfy each rule.

**University Case Study:** Our university case study is a policy that controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. 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 Case Study:** Our health care case study is a policy that controls access by nurses, doctors, patients, and agents (e.g., a patient's spouse) to electronic health records (HRs) and HR items (i.e., entries in health records). 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 Case Study:** Our project management case study is a policy that controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. 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 explanation is similar to the above explanation for the health care case study. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

**Online Video Case Study:** Our online video case study is a policy that controls access to videos by users of an online video service. The generated policy is identical to the "original" ABAC policy presented in Section 10 in the Supplemental Material. That is not surprising, because the "original" policy was actually produced by applying our algorithm to a simple-minded ACL-like policy, which contains, for each combination of video type and rating, one rule specifying the membership categories and age groups permitted to view those videos. The generated policy is much more concise than the ACL-like policy (WSC=20 *vs.* WSC=39) and generalizes better. For example, if a new rating, such as PG-13, is introduced, the minimized policy automatically grants adults permission to view movies with the new rating; the ACL-like policy does not.

**Running Time on Synthetic Attribute Data:** For the first three case studies, we generated a series of synthetic datasets, parameterized by a number $N$, which is the number of departments for the university and project management case studies, and the number of wards for the health care case study. The generated attribute data for users and resources associated with each department or ward are similar to but more numerous than the attribute data in the manually written datasets. For all three case studies, the measured running time is a roughly quadratic function of $N$. Detailed results appear in Figure 15 in Section 10 in the Supplemental Material.

### 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 are less susceptible to bias than hand-written policies. 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. Also, it is more difficult to evaluate the effectiveness of our algorithm on synthetic policies, because when there are differences between the synthetic ABAC policy and the mined ABAC policy (i.e., the ABAC policy mined from the ACL policy generated from the synthetic ABAC policy), it is unclear which policy is better; for example, the synthetic policy might be unnecessarily complicated, and the mined policy might be better. So, 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. Syntactic similarity is in the range [0, 1]; larger values indicate more similarity. The detailed definition of syntactic similarity is in Section 12 in the Supplemental Material. When comparing WSC, we consider the policy mining algorithm to be effective if the mined ABAC policy $Rules_{\mathrm{mined}}$ is simpler (i.e., has lower WSC) than the original synthetic ABAC policy $Rules_{\mathrm{syn}}$. We measure this by the *compression factor*, defined as $\mathrm{WSC}(Rules_{\mathrm{syn}})/\mathrm{WSC}(Rules_{\mathrm{mined}})$. 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 $N_{\mathrm{rule}}$, the desired number of rules, $N_{\mathrm{cnj}}^{\mathrm{min}}$, the minimum number of conjuncts in each attribute expression, and $N_{\mathrm{con}}^{\mathrm{min}}$, the minimum number of constraints in each rule, as an input. 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 case studies or assumed to have a simple functional form (e.g., uniform distribution). For example, the distribution of the number of conjuncts in each attribute expression is based loosely on our case studies 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 case studies and ranges from $N_{\rm con}^{\rm min}$ to $N_{\rm con}^{\rm min} + 2$, 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 both fixed at $N_{\rm attr} = 8$ (for the datasets presented in [13], the maximum number of attributes relevant to access control is 8). Our synthesis algorithm adopts a simple type system, with $N_{\rm type} = 3$ different types. The type of each user attribute and resource attribute is chosen randomly. Let $T(a)$ denote the type of attribute $a$. Each constraint $f$ generated by the algorithm satisfies the condition that $T({\rm uAttr}(f)) = T({\rm rAttr}(f))$. For each rule $\rho$, 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 $\rho$ or can be made to satisfy $\rho$ by appropriate choice of values for attributes with unknown values (i.e., $\perp$). If the count is less than $N_{\rm urp}$, the algorithm generates additional users and resources that together satisfy $\rho$. 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 $\rho$, for each conjunct $e_u(a_u)$ in the UAE $e_u$ in $\rho$, the algorithm generates a user $u'$ by copying an existing user $u$ that (together with some resource) satisfies $\rho$ 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. For each user and resource, the algorithm initially assigns values only to the attributes needed to ensure that the user or resource satisfies the rule under consideration. This may leave several attributes equal to $\perp$. To make the attribute data more realistic, a final step of the algorithm iterates over all users and, for each user $u$, assigns random values to $\min(n_u, 2)$ randomly selected attributes whose value was $\perp$, where $n_u$ is the number of attributes that were equal to $\perp$ for user $u$.

**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 10, with $N_{\rm cnj}^{\rm min}$ ranging from 4 to 0, and with $N_{\rm con}^{\rm min} = 0$. 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 10 synthetic policies and average the results. 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 $N_{\rm con}^{\rm min}$ decreases from 4 to 0, $|UP|$ increases from 1043 to 55895, the syntactic similarity increases from 0.85 to 0.91, and the compression factor decreases from 1.24 to 1.01. Detailed results from some of the experiments appear in Table 17 in Section 13 in the Supplemental Material.

**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 10, with $N_{\rm con}^{\rm min}$ ranging from 2 to 0, and $N_{\rm cnj}^{\rm min} = 0$. For each value of $N_{\rm rule}$, policies with smaller $N_{\rm con}^{\rm min}$ are obtained by removing constraints from synthetic policies with larger $N_{\rm con}^{\rm min}$. 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 $N_{\rm con}^{\rm min}$ 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 $N_{\rm con}^{\rm min}$ decreases from 2 to 0, $|UP|$ increases from 4832 to 26472, the syntactic similarity decreases from 0.81 to 0.78, and the compression factor decreases from 1.11 to 0.8. Detailed results from some of the experiments appear in Table 18 in the Supplemental Material.

**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 $\rho_1$ and $\rho_2$ 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 $\rho$, with probability $P_{over}$ the algorithm generates another rule

$\rho'$ obtained from $\rho$ by randomly removing one conjunct from $\mathrm{uae}(\rho)$ and adding one conjunct (generated in the usual way) to $\mathrm{rae}(\rho)$; typically, $\rho$ and $\rho'$ have a significant amount of overlap. We also add users and resources that together satisfy $\rho'$, so that $[\![\rho']\!] \not\subseteq [\![\rho]\!]$, otherwise $\rho'$ is redundant. This construction is based on a pattern that occurs a few times in our case studies. We generated synthetic policies with sizes ranging from 10 to 50 rules in steps of 10, using the extended algorithm described above. For each value of $N_{\rm rule}$, we generated synthetic policies with $P_{over}$ ranging from 0 to 1 in steps of 0.25, and with $N_{\rm cnj}^{\rm min}$ = 2 and $N_{\rm con}^{\rm min}$ = 0. For each value of $N_{\rm rule}$, as $P_{over}$ 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 as the policies get more similar, the compression factor must get closer to 1). For example, for $N_{\rm rule} = 30$, as $P_{over}$ increases from 0 to 1, the syntactic similarity decreases from 0.87 to 0.84, and the compression factor increases from 1.21 to 2.42. Detailed results from some of the experiments appear in Table 19 in the Supplemental Material.

### 5.3 Generalization

A potential concern with optimization-based policy mining algorithms is that the resulting 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 [10]. The inputs to the methodology are an ABAC policy mining algorithm, an ABAC policy $\pi$, 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 $\pi$ for fraction $f$. Given a set $U'$ of users and a policy $\pi$, the associated resources for $U'$ are the resources $r$ such that $\pi$ 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 $\pi$ 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 0.05 to 0.5 in steps of 0.05 for the university, healthcare, and project management case studies with $N_{\rm dept} = 40$ and $N_{\rm ward} = 40$. For the university and healthcare case studies, the generalization error is 0.13 and 0.15, respectively, at $f = 0.05$, drops to 0.002 at $f = 0.1$, and is zero for $f \geq 0.15$. For the project management case study, the generalization error is 0.22 at $f = 0.05$, drops roughly linearly to 0.01 at $f = 0.2$, and remains at 0.01 through $f = 0.5$. The remaining generalization error is due to one rule (the fifth rule in Figure 13 in the Supplemental Material) that does not get reconstructed; it is difficult to reconstruct because it is similar to and overlaps with another rule (the sixth rule in the same figure).

### 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 an ACL policy, added noise, and applied our policy mining algorithm to the resulting policy. To add a specified level $\nu$ 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 [12, Table 1]. The over-assignments are user-permission tuples generated uniformly at random. The under-assignments are removals of randomly selected user-permission tuples. For each noise level, we ran our policy mining algorithm with noise detection inside a loop that searched for the best values of $\alpha$ (between $0.01$ and $0.1$ in steps of .01) and $\tau$ (between 1 and 10 in steps of 1), because we expect $\tau$ 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 $\alpha$ and $\tau$ are the ones that maximize the Jaccard similarity of the actual (injected) noise and the reported noise. ROC curves that illustrate the tradeoff between false positives and false negatives when tuning the values of $\alpha$ and $\tau$ appear in Section 14 in the Supplemental Material.

We started with the university, health care, and project management case studies with synthetic attribute data with $N_{\rm dept} = 6$ and $N_{\rm ward} = 6$ (we also did some experiments with larger policy instances and got similar results), and with synthetic policies with $N_{\rm rule} = 20$. Figure 8 shows the Jaccard similarity of the actual and reported over-assignments and the Jaccard similarity of the actual and reported under-assignments. Each data point is an average over 10 policies, and error bars show standard deviation. We truncate the error bars at the top of the graph; the untruncated error bar above each point would be the same length as the error bar below the point. Over-assignment detection is quite accurate, always above 0.98 for the case studies and above 0.85 for the synthetic policies. Under-assignment detection is not quite as accurate, always above 0.95 for university and project management, above 0.82 for healthcare, and above 0.64 for the synthetic policies. Intuitively, detecting over-assignments is somewhat easier,

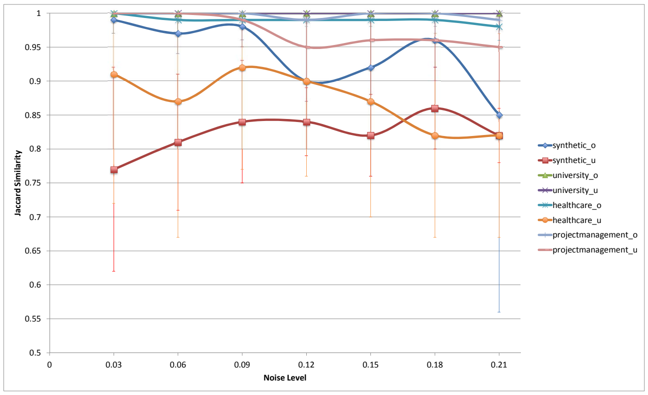


Fig. 8. Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level. Curve names ending with $\_o$ and $\_u$ are for over-assignments and under-assignments, respectively. The university_o curve is almost the same as, and is mostly covered by, the healthcare_o curve.

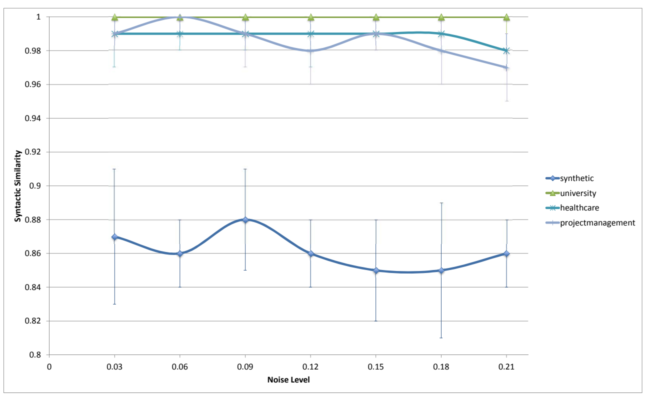


Fig. 9. Syntactic similarity of the original policy and the mined policy, as a function of permission noise level.

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 misclassification of under-assignments. The detailed shapes of the curves are not significant, because the fluctuations are comparable to the standard deviations. The similarities generally trend slightly downward with increasing noise level, although less than we expected. Figure 9 shows the syntactic similarity of the original and mined policies. It is always above 0.97 for the case studies, and above 0.85 for the synthetic policies. The Jaccard similarities and syntactic similarities are generally lower for synthetic policies than case studies, 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 $\nu$, we added 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 (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 15 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.

### 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 case studies and synthetic policies. Details of the translation appear in Section 16 in the Supplemental Material. Progol mostly succeeds in reconstructing the policies for university, on-line video, and project management, except it fails to learn rules with conjuncts or operation sets containing multiple constants, instead producing multiple rules. For example, Progol generates four rules (one for each combination of the constants) corresponding to the rule $\langle$`true`, `type` $\in$ {`schedule`, `budget`}, {`read`, `write`}, `projectsLed` $\ni$ `project`$\rangle$ in the project management case study. In addition, Progol fails to reconstruct two rules in the healthcare case study. 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 $N_{\rm rule} = 5$, our algorithm achieves a compression ratio of 0.52, compared to 0.60 for Progol.

Progol is much slower than our algorithm. For the university, healthcare, and project management case studies with the largest synthetic attribute data used in Figure 15, 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., [8], [16], [17]. Algorithms for mining meaningful RBAC policies from ACLs and user attribute data [8], [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 $\mathrm{uid}$) 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 case studies 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 extensions of their case studies do not have these limitations. In our case studies, 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% in their case studies).

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [11]. 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. Perhaps 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. Another significant difference is that their work does not consider anything analogous to constraints. Also, their work does not consider the size or readability of the generated policy; the classifiers they learn are represented as SVMs. Exploring ABAC policy mining algorithms based on machine learning algorithms is a direction for future work. The main challenge will be effective minimization of WSC, because common machine learning algorithms are not designed to minimize the size of the classifier.

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. Our under-assignment detection follows a similar principle. 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 case studies and synthetic policies demonstrate the algorithm's effectiveness. Directions for future work include optimizing the algorithm, and extending the policy language and algorithm to support arithmetic inequalities and negation in attribute expressions and constraints and to support additional relations among sets in constraints (e.g., $a_{\mathrm{u,m}} \subseteq a_{\mathrm{r,m}}$, as mentioned in Section 2), and exploring ABAC policy mining algorithms based on machine learning algorithms.

## 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] X. Jin, R. Krishnan, and R. Sandhu, "A unified attribute-based access control model covering DAC, MAC and RBAC," *Proc. 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2012)*, pp. 41–55, 2012.

[8] 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.

[9] 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.

[10] 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*, 2009, pp. 101–111.

[11] 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.

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

[13] 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.

[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.

[22] E. Yuan and J. Tong, "Attributed based access control (ABAC) for web services," in *Proc. 2005 IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, 2005, pp. 561–569. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2005.25

**Zhongyuan Xu** Zhongyuan Xu received the B.S. and M.S. degrees in Computer Science from Peking University in 2006 and 2009, respectively. He is currently a Ph.D. candidate at Stony Brook University and 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) [9] 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 $\pi_{\mathrm{RBAC}}$ 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 [9]: 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 $\pi_{\mathrm{RBAC}}$ has minimum number of edges among RBAC policies satisfying this condition. NP-hardness of Edge RMP follows from Theorem 1 in [8], 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 $d_{\mathrm{u}}$ defined by $d_{\mathrm{u}}(u, \mathrm{uid}) = u$, resource attribute data $d_{\mathrm{r}}$ defined by $d_{\mathrm{r}}(r, \mathrm{rid}) = r$, and policy quality metric $Q_{\mathrm{pol}}$ 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 $\pi_{\mathrm{ABAC}}$ to the above ABAC policy mining instance by interpreting each rule as a role. Note that rules in $\pi_{\mathrm{ABAC}}$ 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 $\pi_{\mathrm{ABAC}}$ 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 $\rho$ in $\pi_{\mathrm{RBAC}}$ equals the number of users that satisfy $\rho$ plus the number of resources (i.e., permissions) that satisfy $\rho$. 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 $O(|A_{\mathrm{u}}| \times lc^2_{\mathrm{u,m}} \times |\mathrm{Val_s}|)$. Checking validity of a rule $\rho$ takes time linear in $|[\![\rho]\!]|$. Let $lm$ (mnemonic for "largest meaning") denote the maximum value of $|[\![\rho]\!]|$ among all rules $\rho$ 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 $\rho$ 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 Case Study Details

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

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".

Figure 7 in Section 5.1 contains information about the size of the manually written attribute datasets for the case studies. Figure 10 contains information about the sizes of the synthetic attribute datasets for the case studies, for selected values of $N$. Each row contains the average over 10 synthetic policies with the specified $N$. The standard deviations are negligible.

**University Case Study:** In our university case study, 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 11. 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.

| Case Study | $N$ | $\|U\|$ | $\|R\|$ | $\|UP\|$ | $\|\widehat{[\![\rho]\!]}\|$ |
|---|---|---|---|---|---|
| university | 10 | 314 | 630 | 3699 | 370 |
| healthcare | 10 | 200 | 720 | 1549 | 197 |
| healthcare | 20 | 400 | 1440 | 3151 | 399 |
| project management | 10 | 100 | 200 | 960 | 96 |
| project management | 20 | 200 | 400 | 1920 | 193 |

Fig. 10. Sizes of synthetic attribute datasets for the case studies, for selected values of $N$. $|\widehat{[\![\rho]\!]}|$ is the average number of user-permission tuples that satisfy each rule.

The manually written attribute dataset for this case study contains a few instances of each type of user and resource: two academic departments, a few faculty, a few gradebooks, several students, etc. We generated a series of synthetic 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 example, the synthetic datasets contain 20 students, 5 faculty, and 10 courses per academic department. The amount of data generated for each department is determined by the following parameters: number of applicants per academic department = 5, number of students per academic department = 20, number of faculty per academic department = 5, number of courses per academic department = 10, number of courses taught per faculty = 2, number of courses taken per student = 4, number of staff per administrative department (admissions office and registrar) = 2.

**Health Care Case Study:** In our health care case study, 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 12.

The manually written attribute dataset for this case study 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 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 example, the synthetic datasets contain 10 patients and 4 nurses per ward. and 2 doctors, but that's not more numerous. The amount of data generated for each ward is determined by the following parameters: number of patients per ward = 10, number of HR items per patient = 4, number of topics per HR item = 1, number of nurses

```
// 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. 11. University case study.

```
// 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. 12. Health care case study.

per ward = 4, number of doctors per ward = 2, number of agents per ward = 2, number of patients for which each agent is an agent = 1, number of medical teams per ward = 2, number of medical teams each doctor is on = 2, number of medical areas (used for specialties and topics) = 10, number of medical specialties per doctor = 1.

**Project Management Case Study:** In our project management case study, 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 13.

The manually written attribute dataset for this case study contains a small number of instances of each type of user (managers, accountants, coders, etc.) and each type of resource (two departments, two projects per department, three tasks per project, etc.). We generated a series of synthetic 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 the userAttrib and resourceAttrib statements in the manually written dataset. The amount of data generated for each department is determined by the following parameters: number of projects per department = 2, number of managers per department = 1, number of accountants per department = 1, number of auditors per department = 1, number of planners per department = 1, number of project leaders per department = 2, number of areas of expertise = 2, number of non-employees per expertise area per department = 1, number of employees per expertise area per department = 1, number of budgets per project = 1, number of schedules per project = 1, and for each project, for each area of expertise, for proprietary in {true,false}, there is one assigned task and one unassigned task.

**Online Video Case Study:** Our online video case study is based on the policy in [22], where it is presented as an example of a policy that can be expressed concisely using ABAC but cannot be expressed concisely using RBAC. We modified the policy to use age groups instead of numeric ages. The policy has a more combinatorial character than our other case studies, since permissions depend on combinations of values of multiple user and resource attributes, but not on constraints relating the values of those attributes. User attributes include ageGroup (child, teen, or adult) and memberType (regular or premium). Every resource is a video. Resource attributes include rating (G, PG-13, or R) and videoType (old or new). The policy rules and illustrative userAttrib and resourceAttrib statements appear in Figure 14.

Figure 15 shows the algorithm's running time as a function of $N$ for the university, health care, and project management case studies with synthetic attribute datasets, as described in Section 5.1. Each data point is an average of the running times on 10 sets of synthetic attribute data. Error bars (too short to be seen in some cases) show standard deviation.

```
// 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. 13. Project management case study.

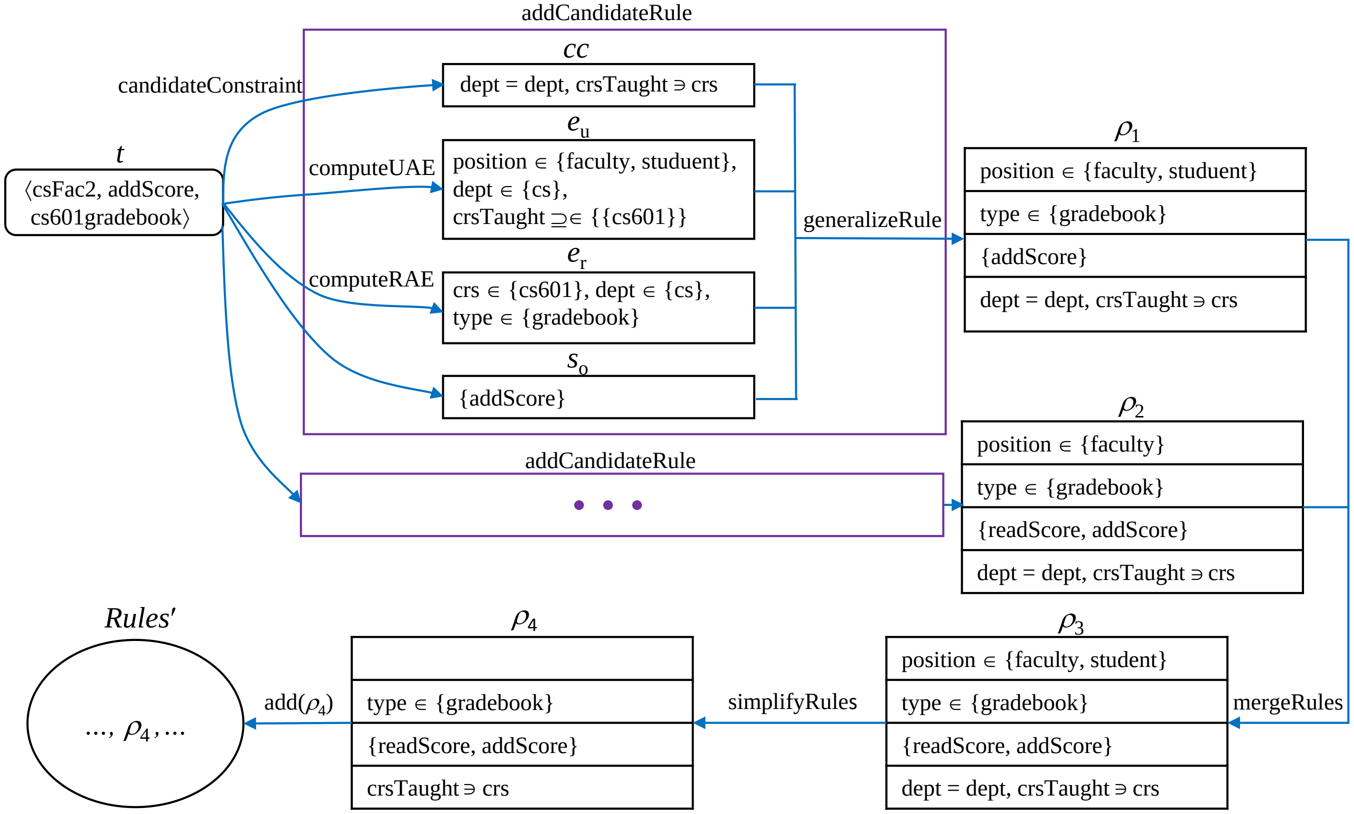


Fig. 16. Diagram representing the processing of one user-permission tuple selected as a seed, in the university case study. Rules are depicted as rectangles with four compartments, corresponding to the four components of a rule tuple.

## 11 Example: Processing of a User-Permission Tuple

Figure 16 illustrates the processing of the user-permission tuple $t = \langle \mathrm{csFac2}, \mathrm{addScore}, \mathrm{cs601gradebook}\rangle$ selected as a seed, in a smaller version of the university case study containing only one rule, namely, the second rule in Figure 11. Attribute data for user csFac2 and resource cs601gradebook appear in Figure 11.

The edge from $t$ to $cc$ labeled "candidateConstraint" represents the call to candidateConstraint, which returns the set of atomic constraints that hold between csFac2 and cs601gradebook; these constraints are shown in the box labeled $cc$. The two boxes labeled "addCandidateRule" represent the two calls to addCandidateRule. Internal details are shown for the first call but elided for the second call. The edges from $t$ to $e_{\mathrm{u}}$ and from $t$ to $e_{\mathrm{r}}$ represent the calls in addCandidateRule to computeUAE and computeRAE, respectively. The call to computeUAE returns a user-attribute expression $e_{\mathrm{u}}$ that characterizes the set $s_{\mathrm{u}}$ containing users $u'$ with permission $\langle \mathrm{addScore}, \mathrm{cs601gradebook}\rangle$ and such that $\mathrm{candidateConstraint}(\mathrm{cs601gradebook}, u') = cc$. The call to computeRAE returns a resource-attribute expression that characterizes $\{\mathrm{cs601gradebook}\}$. The set of operations considered in this call to addCandidateRule is simply $s_{\mathrm{o}} = \{\mathrm{addScore}\}$. The call to generalizeRule generates a candidate rule $\rho_1$ by assigning $e_{\mathrm{u}}$, $e_{\mathrm{r}}$ and $s_{\mathrm{o}}$ to the first three components of $\rho_1$, and adding the two atomic constraints in $cc$ to $\rho_1$ and eliminating the conjuncts in $e_{\mathrm{u}}$ and $e_{\mathrm{r}}$ corresponding to the attributes mentioned in $cc$. Similarly, the second call to addCandidateRule generates another candidate rule $\rho_2$. The call to mergeRules merges $\rho_1$ and $\rho_2$ to form $\rho_3$, which is simplified by the call to simplifyRules to produce a simplified rule $\rho_4$, which is added to candidate rule set $Rules'$.

## 12 Syntactic Similarity

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

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

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 $\rho$ in $Rules$, of the syntactic similarity between $\rho$ 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,

```
// Rules that apply to premium members.
// Premium members of all ages can view movies
// rated G.
rule(memberType=premium; rating in=G; {view}; )
// premium teens can view movies rated PG.
rule(memberType=premium, ageGroup=teen;
     rating=PG; {view}; )
// Premium adults can view movies with all ratings.
rule(memberType=premium, ageGroup=adult; ; {view}; )

// Rules that apply to all member types. These rules
// correspond 1-to-1 with the above rules, transformed
// by dropping the restriction to premium members
// and adding the restriction to old videos.
// Members of all ages can view old movies rated G.
rule(; videoType=old, rating=G; {view}; )
// Teens can view old movies rated PG.
rule(ageGroup=teen; videoType=old, rating=PG; {view}; )
// Adults can view old movies with all ratings.
rule(ageGroup=adult; videoType=old; {view}; )

// An illustrative user attribute statement.
userAttrib(child1r, ageGroup=child,
           memberType=regular)
// An illustrative resource attribute statement.
resourceAttrib(TheLionKing, rating=G, videoType=old)
```

Fig. 14. Online video case study.

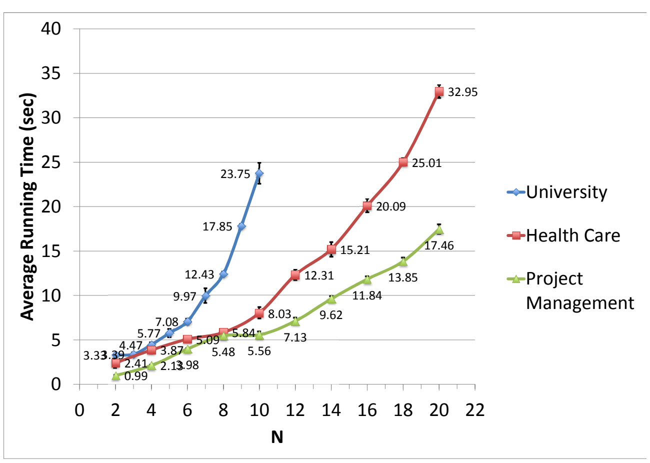


Fig. 15. Running time of the algorithm on synthetic datasets for three case studies. The $x$-axis is $N_{\mathrm{dept}}$ for the university and project management case studies and $N_{\mathrm{ward}}$ for the health care case study.

considered in both orders (this makes the relation symmetric).

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

## 13 Tables of Results from Experiments with Synthetic Policies

Table 17 contains results from some of the experiments with synthetic policies with varying number of conjuncts. The standard deviation in running time is large because the moderately large variation in $|UP|$ gets magnified by the algorithm's super-linear running time. Table 17 contains results from some of the experiments with synthetic policies with varying number of constraints. Table 19 contains results from some of the experiments with synthetic policies with varying overlap between rules.

## 14 ROC Curves for Noise Detection Parameters

When tuning the parameters $\alpha$ and $\tau$ used in noise detection (see Section 4.3), there is a trade-off between true positives and false positives. To illustrate the trade-off, the Receiver Operating Characteristic (ROC) curve in Figure 20 shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on $\alpha$ and $\tau$ for synthetic policies with 20 rules and 6% noise, split between under-assignments and over-assignments as described in Section 5.4. Figure 21 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 $\tau$ 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 $\alpha$, 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 $\alpha$ 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 $\tau$, there is some trade-off between the TPR and FPR. For $\alpha \geq 0.02$, the FPR equals 0 independent of $\tau$, so there is no trade-off: the best values of $\tau$ are the ones with the highest TPR.

| $N_{\text{rule}}$ | $N_{\text{cnj}}^{\text{min}}$ | $\|U\|$ | $\|R\|$ | $\|UP\|$ | $\|\widehat{[\![\rho]\!]}\|$ | Synt. Sim. | Compression | Time |
|---|---|---|---|---|---|---|---|---|
| 10 | 4 | 203 (16) | 63 (4) | 266 (76) | 23 (5) | 0.74 (0.04) | 1.34 (0.14) | 0.60 (0.84) |
| ″ | 2 | ″ | ″ | 619 (85) | 124 (24) | 0.81 (0.03) | 1.29 (0.09) | 0.50 (0.53) |
| ″ | 0 | ″ | ″ | 1754 (1467) | 384 (173) | 0.90 (0.05) | 1.13 (0.11) | 1.70 (1.49) |
| 50 | 4 | 1034 (17) | 310 (9) | 1043 (145) | 20 (3) | 0.85 (0.03) | 1.24 (0.06) | 1.30 (0.95) |
| ″ | 2 | ″ | ″ | 2810 (228) | 112 (11) | 0.89 (0.02) | 1.19 (0.05) | 2.10 (2.13) |
| ″ | 0 | ″ | ″ | 55895 (104492) | 1459 (2497) | 0.91 (0.03) | 1.01 (0.13) | 59.90 (105.48) |

Fig. 17. Experimental results for synthetic policies with varying $N_{\text{cnj}}^{\text{min}}$. “Synt. Sim.” is syntactic similarity. “Compression” is the compression factor. Standard deviations are shown in parentheses.

| $N_{\text{rule}}$ | $N_{\text{con}}^{\text{min}}$ | $\|U\|$ | $\|R\|$ | $\|UP\|$ | $\|\widehat{[\![\rho]\!]}\|$ | Synt. Sim. | Compression | Time |
|---|---|---|---|---|---|---|---|---|
| 10 | 2 | 130 (18) | 45 (5) | 608 (165) | 57 (14) | 0.77 (0.05) | 1.18 (0.15) | 1.50 (1.43) |
| ″ | 1 | ″ | ″ | 896 (235) | 128 (26) | 0.76 (0.05) | 1.05 (0.20) | 1.30 (1.49) |
| ″ | 0 | ″ | ″ | 1493 (502) | 229 (61) | 0.74 (0.04) | 0.85 (0.16) | 2.70 (2.58) |
| 50 | 2 | 567 (33) | 269 (20) | 4832 (1078) | 94 (22) | 0.81 (0.03) | 1.11 (0.09) | 17.70 (5.48) |
| ″ | 1 | ″ | ″ | 7023 (2064) | 232 (55) | 0.80 (0.03) | 0.96 (0.14) | 30.60 (13.09) |
| ″ | 0 | ″ | ″ | 26472 (12606) | 652 (201) | 0.78 (0.04) | 0.80 (0.12) | 166.00 (148.31) |

Fig. 18. Experimental results for synthetic policies with varying $N_{\text{con}}^{\text{min}}$. Standard deviations are shown in parentheses.

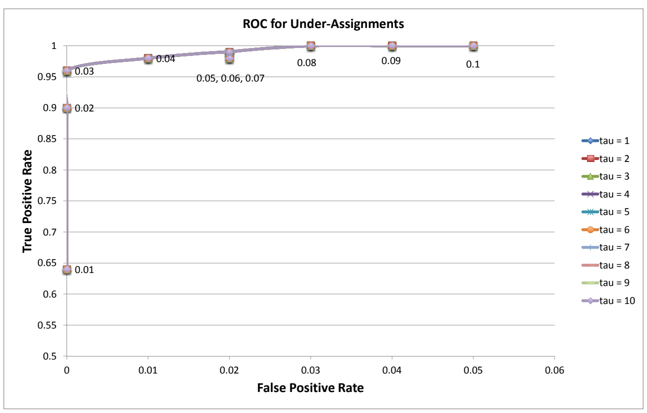


Fig. 20. ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on $\alpha$ and $\tau$.

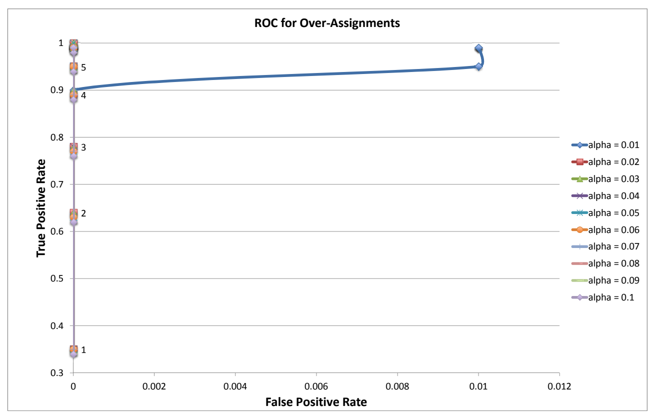


Fig. 21. ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for over-assignments on $\alpha$ and $\tau$.

## 15 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, graphs of Jaccard similarity of actual and reported noise, and syntactic similarity of original and mined policies, appear in Figures 22 and 23, respectively. Each data point is an average over 10 policies, and error bars show standard deviation.

## 16 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 $\text{Val}_{\text{s}}$) 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 $\text{Val}_{\text{m}}$) 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

| $N_{\text{rule}}$ | $P_{over}$ | $\lvert U\rvert$ | $\lvert R\rvert$ | $\lvert UP\rvert$ | $\lvert\widehat{[\![\rho]\!]}\rvert$ | Synt. Sim. | Compression | Time |
|---|---|---|---|---|---|---|---|---|
| 30 | 0.00 | 431 (33) | 132 (10) | 1751 (146) | 57 (6) | 0.87 (0.02) | 1.21 (0.04) | 0.30 (0.95) |
| ″ | 0.25 | 457 (51) | 133 (10) | 1747 (250) | 112 (13) | 0.85 (0.03) | 1.59 (0.62) | 0.10 (0.32) |
| ″ | 0.50 | 468 (41) | 133 (11) | 1889 (334) | 165 (20) | 0.85 (0.04) | 1.62 (0.57) | 0.80 (1.87) |
| ″ | 0.75 | 475 (26) | 136 (8) | 1823 (335) | 221 (24) | 0.83 (0.04) | 1.82 (0.58) | 0.50 (0.53) |
| ″ | 1.00 | 514 (32) | 143 (13) | 1638 (267) | 270 (14) | 0.84 (0.04) | 2.42 (0.23) | 0.10 (0.32) |

Fig. 19. Experimental results for synthetic policies with varying $P_{over}$. Standard deviations are shown in parentheses. $\lvert\widehat{[\![\rho]\!]}\rvert$ is the average number of user-permission tuples that satisfy each rule.

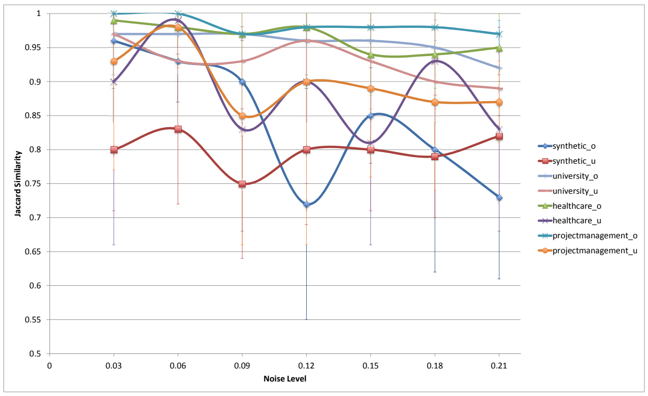


Fig. 22. Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level due to permission noise and attribute noise.

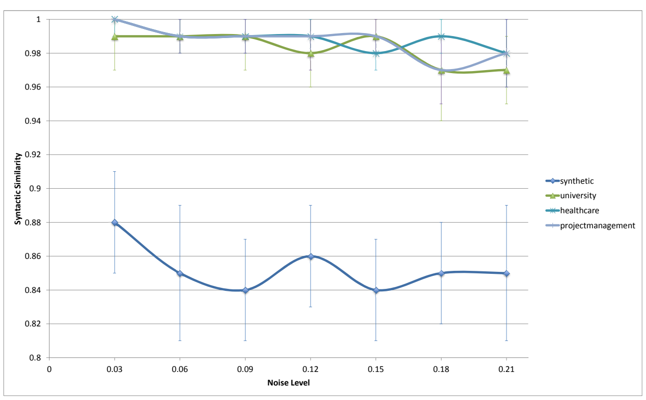


Fig. 23. Syntactic similarity of the original policy and the mined policy, as a function of permission noise level due to permission noise and attribute noise.

`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,` $a$`U(+user, #`$a$`Type))`. 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 `-`$a$`Type` instead of `#`$a$`Type`, 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,` $a$`U_equals_`$\bar{a}$`R(+user,+resource));` the predicate $a$U_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 $a$U($u,v$) where $v = d_{\mathrm{u}}(u,a)$. For each user $u$ and each multi-valued user attribute $a$, we generate a statement $a$U($u,v$) 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_āR(U,R) :- aU(U,X), āR(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_āR` 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.