Mining Attribute-Based Access Control Policies from RBAC Policies

Zhongyuan Xu
Computer Science Department
Stony Brook University
Stony Brook, USA

Scott D. Stoller
Computer Science Department
Stony Brook University
Stony Brook, USA

***Abstract*—Role-based access control (RBAC) is very widely used but has notable limitations, prompting a shift towards attribute-based access control (ABAC). However, the cost of developing an ABAC policy can be a significant obstacle to migration from RBAC to ABAC. This paper presents the first formal definition of the problem of mining ABAC policies from RBAC policies and attribute data, and the first algorithm specifically designed to mine an ABAC policy from an RBAC policy and attribute data.**

Keywords: role mining; role-based access control; attribute-based access control;

1. INTRODUCTION

Role-based access control (RBAC) [1] is very widely used but has notable limitations, prompting a shift towards attribute-based access control (ABAC) [2], which allows policies to be written in a more flexible and higher-level way. However, the cost of developing an ABAC policy can be a significant obstacle to migration from RBAC to ABAC. Policy mining algorithms can significantly reduce this cost, by partially automating the construction of an ABAC policy from an RBAC policy with accompanying data about attributes of users and resources.

The main contributions of this paper are (1) the first formal definition of the problem of mining ABAC policies from RBAC policies and attribute data, and (2) the first algorithm specifically designed to mine an ABAC policy from an RBAC policy and attribute data.

An important feature of our problem definition is that it requires that some aspects of the structure of the RBAC policy be preserved in the ABAC policy. This is important, because the structure of the RBAC policy may reflect expert design decisions by the policy author.

To evaluate the effectiveness of our algorithm at producing intuitive, high-level ABAC policies from RBAC policies, we manually wrote case study policies in RBAC and ABAC, applied our algorithm to the RBAC policy and accompanying attribute data, and compared the generated ABAC policy to the manually written one. Our algorithm successfully generates ABAC policies identical or similar to the manually written ABAC policies. The user can optionally supply some guidance to our algorithm, by indicating that some attributes are important. In our case studies, appropriate guidance can easily be determined based on the obvious importance of some attributes, or from examination of the policy generated with no guidance. With a small amount of such guidance, our algorithm generates ABAC policies identical or very similar to the manually written ones.

In practice, the available attribute data is often incomplete. To evaluate the effectiveness of our algorithm in such cases, we also performed experiments in which we omitted some relevant attribute data, and demonstrated that our algorithm uses role membership information effectively as a substitute for missing attribute data.

To demonstrate the significance of preserving the structure of the RBAC policy, we wrote variants of some RBAC policies, with the same semantics (i.e., same user-permission relation) but different structure (i.e., different roles), and showed that our algorithm generates a different ABAC policy with corresponding structure for each variant.

2. RBAC POLICY LANGUAGE

An *RBAC policy* is a tuple $\langle U, Res, Op, Roles, UA, PA, RH\rangle$, where U is a set of users, Res is a set of resources, Op is a set of operations, $Roles$ is a set of roles, $UA \subseteq U \times Roles$ is the user-role assignment, $PA \subseteq Roles \times Perm$ is the permission-role assignment, and the role hierarchy RH is an acyclic transitive binary relation on roles. A *permission* is a pair containing a resource and an operation, and $Perm = Res \times Op$. A tuple $\langle r, r'\rangle$ in RH means that r is junior to r' (or, equivalently, r' is senior to r). This means that r inherits members from r', and r' inherits permissions from r. The *authorized users* of a role include the role's directly assigned users and its inherited users. The *authorized permissions* of a role are defined similarly. These ideas are expressed in the equations below.

$$\text{asgndU}(r)=\{u \in U \mid \langle u, r\rangle \in UA\}$$

$$\text{asgndP}(r)=\{p \in Perm \mid \langle r, p\rangle \in PA\}$$

$$\text{ancestors}(r)=\{r' \in Roles \mid \langle r, r'\rangle \in RH\}$$

$$\text{descendants}(r)=\{r' \in Roles \mid \langle r', r\rangle \in RH\}$$

$\text{authU}(r) = \text{asgndU}(r) \cup \bigcup_{r' \in ancestors(r)} \text{asgndU}(r')$

$\text{authP}(r) = \text{asgndP}(r) \cup \bigcup_{r' \in descendants(r)} \text{asgndP}(r')$

The user-permission assignment induced by a role r and an RBAC policy π with the above form are defined by $\text{authUP}(r) = \text{authU}(r) \times \text{authP}(r)$ and $M(\pi) = \bigcup_{r \in Roles} \text{authUP}(r)$, respectively.

3. ABAC POLICY LANGUAGE

ABAC policies refer to attributes of users and resources. Given a set U of users and a set A_u of user attributes, user attribute data is represented by a function d_u such that $d_u(u, a)$ is the value of attribute a for user u. There is a distinguished user attribute uid that has a unique value for each user. Similarly, given a set *Res* of resources and a set A_r of resource attributes, resource attribute data is represented by a function d_r such that $d_r(r, a)$ is the value of attribute a for resource r. There is a distinguished resource attribute rid that has a unique value for each resource. Let Val_s be the set of possible atomic (i.e., non-set) values of attributes. We assume Val_s includes a distinguished value $\perp$ used to indicate that an attribute's value is unknown (or irrelevant). We assume the set A_u of user attributes can be partitioned into a set $A_{u,1}$ of *single-valued user attributes*, whose values are in Val_s, and a set $A_{u,m}$ of *multi-valued user attributes*, whose values are in $\text{Val}_m = \text{Set}(\text{Val}_s \backslash \{\perp\}) \cup \perp$, where $\text{Set}(S)$ is the powerset of set S. Similarly, we assume the set A_r of resource attributes can be partitioned into a set $A_{r,1}$ of *single-valued resource attributes* and a set $A_{r,m}$ of *multi-valued resource attributes*.

Attribute expressions are used to express the sets of users and resources associated with rules. A *user-attribute expression* (UAE) e is a function such that, for each user attribute a, $e(a)$ is either a set (interpreted as a disjunction) of possible values of a excluding $\perp$ (i.e., a subset of $\text{Val}_s \backslash \{\perp\}$ or $\text{Val}_m \backslash \{\perp\}$, depending on whether a is single-valued or multi-valued) or $\top$. The symbol $\top$ indicates that the expression imposes no constraint on the value of the attribute. We refer to the set $e(a)$ as the *conjunct* for attribute a. We say that expression e *uses* an attribute a if $e(a) \neq \top$. Let $\text{attr}(e)$ denote the set of attributes used by e. Let $\text{attr}_1(e)$ and $\text{attr}_m(e)$ denote the sets of single-valued and multi-valued attributes, respectively, used by e.

A user u *satisfies* a user-attribute expression e, denoted $u \models e$, iff $(\forall a \in A_{u,1}.\ e(a) = \top \vee \exists v \in e(a).\ d_u(u, a) = v)$ and $(\forall a \in A_{u,m}.\ e(a) = \top \vee \exists v \in e(a).\ d_u(u, a) \supseteq v)$. For multi-valued attributes, we use the condition $d_u(u, a) \supseteq v$ instead of $d_u(u, a) = v$ because elements of a multi-valued user attribute typically represent some type of capabilities of a user, so using $\supseteq$ expresses that the user has the specified capabilities (and possibly more).

For example, if $A_u = \{\text{dept, position}\}$, the function e with $e(\text{dept}) = \{\text{CS}\}$ and $e(\text{position}) = \{\text{grad, ugrad}\}$ and $e(\text{courses}) = \{\{\text{CS101, CS102}\}\}$ 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 use in examples. 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 supseteqIn v (indicating that a is a superset of a set in 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 may be written as dept = CS $\wedge$ position $\in$ {ugrad, grad} $\wedge$ courses $\supseteq$ {CS101, CS102}.

The *meaning* of a user-attribute expression e, denoted $M_u(e)$ is the set of users in U that satisfy it: $M_u(e) = \{u \in U \mid u \models e\}$. User attribute data is an implicit argument to $M_u(e)$. We say that *e characterizes* the set $M_u(e)$.

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

An *atomic constraint* is a formula f of the form $a_{u,m} \supseteq a_{r,m}$, $a_{u,m} \ni a_{r,1}$, or $a_{u,1} = a_{r,1}$, where $a_{u,1} \in A_{u,1}$, $a_{u,m} \in A_{u,m}$, $a_{r,1} \in A_{r,1}$, and $a_{r,m} \in A_{r,m}$. The first two forms express that user attributes contain specified values. This is a common type of constraint, because user attributes typically represent some type of capabilities of a user. Other forms of atomic constraint are possible (e.g., $a_{u,m} \subseteq a_{r,m}$) but less common, so we do not consider them in this paper. Let $\text{uAttr}(f)$ and $\text{rAttr}(f)$ refer to the user attribute and resource attribute, respectively, used in f. User u and resource r satisfy an atomic constraint f, denoted $\langle u, r \rangle \models f$, if $d_u(u, \text{uAttr}(f)) \neq \perp$ and $d_r(r, \text{rAttr}(f)) \neq \perp$ and formula f holds when the values $d_u(u, \text{uAttr}(f))$ and $d_r(r, \text{rAttr}(f))$ are substituted in it.

A *constraint* is a set (interpreted as a conjunction) of atomic constraints. User u and resource r *satisfy* a constraint c, denoted $\langle u, r \rangle \models c$, if they satisfy every atomic constraint in c.

A *user-permission* tuple is a pair $\langle u, \langle r, o \rangle \rangle$ containing a user and a permission. As in RBAC, a permission is a pair containing a resource and an operation. A *user-permission relation* is a set containing such tuples.

A *rule* is a tuple $\langle e_u, e_r, O, c \rangle$, where e_u is a user-attribute expression, e_r is a resource-attribute expression, O is a set of operations, and c is a constraint. For a rule $\rho = \langle e_u, e_r, O, c \rangle$, let $\text{uae}(\rho) = e_u$, $\text{rae}(\rho) = e_r$, $\text{ops}(\rho) = O$, and $\text{con}(\rho) = c$. User u and permission $\langle r, o \rangle$ *satisfy* a rule ρ, denoted $\langle u, \langle r, o \rangle \rangle \models \rho$, if $u \models \text{uae}(\rho) \wedge r \models \text{rae}(\rho) \wedge o \in \text{ops}(\rho) \wedge \langle u, r \rangle \models \text{con}(\rho)$.

An *ABAC policy* is a tuple $\langle U, Res, Op, A_u, A_r, d_u, d_r, Rules \rangle$ where U, Res, A_u, A_r, d_u and d_r are as described above, Op is a set of operations, and $Rules$ is a set of rules.

The user-permission relation induced by a rule ρ is $M(\rho) = \{\langle u, \langle r, o \rangle \rangle \in U \times Res \times Op \mid \langle u, \langle r, o \rangle \rangle \models \rho\}$. Note that U, Res,

d_u and d_r are implicit arguments to M(ρ).

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

4. PROBLEM DEFINITION

An RBAC policy π_{RBAC} is *semantically consistent* with an ABAC policy π if $M(\pi_{RBAC}) = M(\pi)$.

Our goal is to mine an ABAC policy that is semantically consistent with a given RBAC policy and preserves the structure of the RBAC policy. A first thought is to require a 1-to-1 correspondence between roles and rules; in other words, for each role *r*, the mined policy contains a rule that covers the same user-permission tuples. However, this requirement is too strict, for two reasons. First, some roles cannot be expressed as a single rule, because the set of permissions granted by a rule must be expressible as the Cartesian product of a set of resources and a set of operations, while the set of permissions granted by a role can be arbitrary (although, in practice, it is often expressible as a Cartesian product). Second, it is often desirable to express multiple related roles by a single rule; for example, a set of roles, each granting certain permissions to staff in a particular department, can be expressed more concisely by a single rule that uses a constraint to ensure that each user is granted permissions appropriate to his or her department. Therefore, we relax this requirement in two ways. First, we split the given roles, so that each role's set of assigned permissions is the Cartesian product of a set of resources and a set of operations, and we require a correspondence between the resulting split roles and the mined rules. Second, we allow multiple roles to correspond to a single rule.

Given a set *P* of permissions, we want to express *P* as a sum (union) of Cartesian products. Let ops(*P*) be the set of operations that appear in *P*. Let resources(o, *P*) be the set of resources associated with *o* in *P*, i.e., $\{r \in Res \mid \langle r, o \rangle \in P\}$. Define two operations to be equivalent if they are associated with the same resources in *P*, i.e., $o \equiv_P o'$ iff resources(*o*, *P*) = resources(*o'*, *P*). Let *S* be a partition of ops(*P*) containing the equivalence classes of *O* with respect to $\equiv_P$. Define $SOP(P) = \bigcup_{O \in S} \{\langle resources(O), O \rangle\}$, where resources(*O*) is the set of resources associated with any operation in *O* (by definition, all operations in *O* are associated with the same resources). Note that $P = \bigcup_{\langle R,O \rangle \in SOP(P)} R \times O$.

Given an RBAC policy π_{RBAC} = ⟨*U*, *Res*, *Op*, *Roles*, *UA*, *PA*, *RH*⟩, the sum-of-products policy SOP(π_{RBAC}) is ⟨*U*, *Res*, *Op*, *Roles'*, *UA'*, *PA'*, *RH'*⟩, where

$Roles' = \bigcup_{r \in Roles} \bigcup_{\{R,O\} \in SOP(\text{asgndP}(r))} \{\langle r, R, O \rangle\}$

$UA' = \bigcup_{\langle r,R,O \rangle \in Roles'} \text{asgndU}(r) \times \{\langle r, R, O \rangle\}$

$PA' = \bigcup_{\langle r,R,O \rangle \in Roles'} \{\langle r, R, O \rangle\} \times (R \times O)$

$RH' = \{\langle r, R, O \rangle, \langle r', R', O' \rangle \in Roles \times Roles' \mid \langle r, r' \rangle \in RH\}$

Note that we use tuples of the form ⟨*r*, *R*, *o*⟩ as role names in the sum-of-products policy. Note that $M(\pi_{RBAC}) = M(SOP(\pi_{RBAC}))$. For a role *r* in a sum-of-products RBAC policy, let $\text{asgndRes}(r) = \bigcup_{\langle r,o \rangle \in \text{asgndP}(r)} \{r\}$ and $\text{asgndOp}(r) = \bigcup_{\langle r,o \rangle \in \text{asgndP}(r)} \{o\}$.

Given an RBAC policy π_{RBAC} = ⟨*U*, *Res*, *Op*, *Roles*, *UA*, *PA*, *RH*⟩ and an ABAC policy π = ⟨*U*, *Res*, *Op*, A_u, A_r, d_u, d_r, *Rules*⟩, a *structural correspondence* between π_{RBAC} and π is an onto function κ from the roles in SOP(π_{RBAC}) whose authUP is non-empty to the rules in π such that, for each rule ρ, $M(\rho) = \bigcup_{r \in \kappa^{-1}(\rho)} \text{authUP}(r)$, where κ^{-1} is the inverse of κ; i.e., $\kappa^{-1}(\rho)$ is the set of roles that map to rule ρ.

An ABAC policy is *structurally consistent* with an RBAC policy if there exists a structural correspondence between them.

Among ABAC policies semantically and structurally consistent with a given RBAC policy RBAC, which ones are preferable? 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. Thus, an initial idea is to require that each of these attributes is used in the ABAC policy only if necessary, i.e., only if every ABAC policy that is semantically and structurally consistent with π_{RBAC} contains rules that use that attribute.

We refine this initial idea as follows. According to this initial idea, uid is used only when the information available from other attributes is insufficient to "explain" parts of the permission assignment, i.e., insufficient to characterize the sets of users that appear in the RBAC policy. In practice, this is likely to occur fairly often, because the available attribute information is often incomplete. However, rules that use uid to enumerate sets of users by their user identifiers are likely to be lower-level and harder to understand than the corresponding parts of the original RBAC policy. Therefore, we prohibit use of uid in the ABAC policy, introduce a user attribute that expresses role membership, and allow this new user attribute to be used (instead of uid) when necessary to achieve semantic and structural consistency with the RBAC policy.

A *policy quality metric* is a function 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 might seem counter-intuitive at first glance but is natural for metrics based on policy size.

The ABAC-from-RBAC policy mining problem is: give an RBAC policy π_{RBAC} = ⟨*U*, *Res*, *Op*, *Roles*, *UA*, *PA*, *RH*⟩,

attribute data $\langle A_u, A_r, d_u, d_r \rangle$, and a policy quality metric Q_{pol}, find a set *Rules* of rules such that the ABAC policy $\pi = \langle U, Res, Op, A_u \cup \{\text{roles}\}, A_r, d'_u, d_r, Rules \rangle$ (1) is semantically and structurally consistent with π_{RBAC}, (2) does not use uid, (3) uses roles and rid only when necessary, and (4) has the best quality, according to Q_{pol}, among policies that satisfy conditions (1) through (3). Here, d'_u is d_u extended with a user attribute "roles" defined by: d'_u (u, roles) = $\{r \in Roles' \mid u \in \text{authU}(r)\}$. For simplicity, we assume roles $\notin A_u$.

For the policy quality metric, we use *weighted structural complexity* [3], a generalization of policy size. The WSC of an ABAC policy is the WSC of the set *Rules* of rules in the policy, defined by

$$\text{WSC}(e) = \sum_{a \in attr_l(e)} |e(a)| + \sum_{a \in attr_m(e), s \in e(a)} |s|$$

$$\text{WSC}(\langle e_u, e_r, O, c \rangle) = w_1\text{WSC}(e_u) + w_2\text{WSC}(e_r) + w_3|\text{O}| + w_4|c|$$

$$\text{WSC}(Rules) = \sum_{\rho \in Rules} \text{WSC}(\rho)$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights. In the experiments in Section 6, all weights equal 1.

5. POLICY MINING ALGORITHM

At a high level, our algorithm works as follows. First, it splits the roles in the given RBAC policy so that each role's assigned permissions are the Cartesian product of a set of resources and a set of operations. Second, it constructs an ABAC policy rule corresponding to each role (the splitting in the first step is necessary to ensure that each role can be translated into a single rule). Finally, it attempts to improve the policy by merging and simplifying rules.

Let the inputs to the algorithm be denoted as in the problem statement. Let $\pi'_{RBAC} = \langle U, Res, Op, Roles', UA', PA', RH' \rangle$ be the sum-of-products policy for RBAC. Top-level pseudo-code for our policy mining algorithm appears in Figure 1. It calls several functions, described next.

The function computeUAE(s, U) computes a user-attribute expression e_u that characterizes the set s of users. Preference is given to attribute expressions that do not use uid, as discussed in Section 4. After constructing a candidate expression e, it calls 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. The expression e_u returned by computeUAE might not be minimum-sized among expressions that characterize s: it is possible that some attributes mapped to a set of values by e_u can instead be mapped to $\top$.

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.

```
// Rules is the set of rules
Rules = Ø
// κ is the structural correspondence
κ = Ø
for r in Roles'
    if authUP(r).isEmpty
        continue
    end if
    // create a rule corresponding to r
    e_u = computeUAE(authU(r))
    e_r = computeRAE(asgndRes(r))
    O = asgndOp(r)
    cc = ⋂_{u∈authU(r), s∈asgndRes(r)} candConstr(u, s)
    ρ = ⟨e_u, e_r, O, cc⟩
    Rules.add(ρ)
    κ.add(⟨r, ρ⟩)
end for
// Rules is semantically and structurally consistent with
// π_RBAC. Try to improve its quality, by repeatedly merging
// and simplifying rules, until this has no effect.
mergeRules(Rules, κ)
while simplifyRules(Rules, κ)
   if not mergeRules(Rules, κ)
      break
   end if
end while
useRoleAttribute(Rules, κ)
return ⟨Rules, κ⟩
```

Figure 1. Top-level pseudocode for policy mining algorithm and computeUAE(s) that computes a user-attribute expression that characterizes set s of users

The function candConstr(u, r), mnemonic for "candidate constraint", returns a set containing all atomic constraints that hold between user u and resource r.

The function mergeRules(*Rules*, κ) attempts to reduce the WSC of *Rules*, while preserving semantic and structural consistency, by removing redundant rules and merging pairs of rules. A rule ρ is subsumed by a role ρ' if $M(\rho) \subseteq M(\rho')$. A rule ρ in *Rules* is redundant if it is subsumed by another rule in *Rules*. Informally, rules ρ_1 and ρ_2 are merged by taking, for each attribute, the union of the conjuncts in ρ_1 and ρ_2 for that attribute. If adding the resulting rule ρ_{merge} and removing rules subsumed by ρ_{merge} (including ρ_1 and ρ_2) preserves structural consistency, then these changes are made to *Rules*, and the structural correspondence κ is updated accordingly. mergeRules(*Rules*, κ) updates *Rules* and κ in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(*Rules*, κ) attempts to simplify the rules in *Rules*. It updates its arguments *Rules* and κ in place, replacing rules in *Rules* with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several ways, which are embodied in the following

simplification functions that it calls. Generally, each of these simplification functions returns a Boolean indicating whether changes were made; this information is used in the top-level pseudocode in Figure 1 to determine whether another iteration of merging and simplification is necessary. The function elimRedundantSets is described above. It returns false, even if some redundant sets were eliminated, because elimination of redundant sets does not affect the meaning or mergeability of rules, so it should not trigger another iteration of merging and simplification. The function elimConjuncts(ρ, *Rules*, κ, *UP*) attempts to increase the quality of rule ρ by eliminating some conjuncts. Based on our primary goal of minimizing the generated policy's WSC, the quality of rule ρ is $|M(\rho)|$ / $WSC(\rho)$. A set of *unremovable attributes* can be specified, containing attributes that should not be eliminated, typically 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. The function elimConstraints(ρ, *Rules*, κ, *UP*) attempts to improve the quality of ρ by removing unnecessary atomic constraints from ρ's constraint. An atomic constraint is *unnecessary* in a rule ρ if removing it from ρ's constraint leaves ρ valid. The function elimElements(ρ, *Rules*, κ, *UP*) attempts to decrease the WSC of rule ρ by removing elements from sets in conjuncts for multi-valued user attributes, if removal of those elements produces a rule ρ' that can replace the rules it subsumes; note that, because $\supseteq$ is used in the semantics of user attribute expressions, the set of user-permission pairs that satisfy a rule is unchanged or increased (never decreased) by such removals.

The function useRoleAttribute(*Rules*, κ) replaces uses of "uid" with uses of the user attribute "roles", which is defined in the policy mining problem definition in Section 4.

6. EVALUATION

We evaluated our algorithm on manually written case studies. Experiments with a real RBAC policy and real attribute data would be better, but unfortunately, we do not have access to such information. The policies are small but non-trivial and realistic. Brief descriptions of the case studies are included here. Full details are available at **http://www.cs.stonybrook.edu/~stoller/abac-from-rbac/**. The ABAC policies for the case studies are similar to those in [4].

6.1 Experiments with Full Attribute Data

These experiments demonstrate that, when all relevant attribute data is available, our algorithm successfully produces an intuitive high-level ABAC policy from an RBAC policy. We manually wrote semantically consistent case study policies in RBAC and ABAC, applied our algorithm to the RBAC policy and accompanying attribute data, and compared the generated ABAC policy with the manually written one.

University Case Study Our university case study is a policy that controls access to applications (for admission), gradebooks, transcripts, and course schedules. There are roles for students in each course, TAs of each course, instructor of each course, chairman of each department, registrar staff, admissions staff, and applicants for admission. The permission assignment allows a student to read his/her transcript, an instructor to assign grades for courses he/she teaches, etc.

Health Care Case Study Our health care case study is a policy that controls access to electronic health records (HRs) and HR items (i.e., entries in health records). There are roles for nurses in each ward (e.g., oncology ward), each medical team, each medical specialty on each medical team (e.g., oncologists on team 1), each patient, and agents for each patient. The permission assignment allows a nurse to add note items in health records for patients in the ward he/she works in, a patient and his/her agents to read note items in the patient's medical record, members of a medical team to read items appropriate to their medical specialty in health records of patients treated by that team, etc.

Project Management Case Study Our project management case study is a policy that controls access to budgets, schedules, and tasks associated with projects. There are roles for the manager of each department; for the accountants, auditors, planners, leaders, designers, and coders working on each project; and for the designers and coders assigned to each task. The roles also distinguish employees from non-employees (contractors). Role hierarchy is used to combine the roles for users of each specialty working on a project into a role for all users working on the project. The permission assignment allows a user working on a project to read the project schedule, a user working on a task to update the status of the task, a non-employee working on a project to read information about non-proprietary tasks in that project that match his/her technical expertise, etc.

For each case study, with no guidance (i.e., no attributes are declared unremovable), the generated ABAC policy is almost identical to the manually written ABAC policy, with a 1-to-1 correspondence between rules in the two policies, and with small differences between some corresponding rules. If resource type is specified as an unremovable attribute, then the generated policy is identical to the manually written ABAC policy for university case study, and the generated policy has only one additional conjunct in one rule for health care and project management case studies (the additional conjunct reduces overlap between rules).

6.2 Experiments with Incomplete Attribute Data

These experiments demonstrate that, when some relevant attribute information is unavailable, our algorithm successfully produces an intuitive high-level ABAC policy that uses the available attribute data and uses role membership information as a substitute for missing attribute data.

For the health care case study, we deleted the user attribute data specifying which users are agents for which patients; this data seems less essential to the hospital's IT system, and hence more likely to be unavailable, than employee-related user attribute data. With this input, the generated ABAC policy is mostly identical to the ABAC policy generated with full attribute data (as described above): rules unrelated to agents are unaffected, while rules granting

permissions to agents are replaced with similar rules that use agent roles instead of the "agent for" attribute. The number of agent-related rules increases, because a separate rule is needed for each patient's agents.

For the university case study, we deleted the user attribute data specifying whether a user is a department chair. As expected, only the rule granting permissions to department chairs is affected, and the only change in that rule is replacement of the conjunct "isChair=true" in the user attribute expression with the conjunct "role supseteqIn {{eeChair}, {csChair}}".

6.3 Experiment with Varying Policy Structure

This experiment demonstrates how the structure of the RBAC policy propagates into the structure of the generated ABAC policy. As a small example, consider two similar RBAC polices π_{R1} and π_{R2}. π_{R1} has three roles: csStudent, eeStudent, and student, where members of csStudent role have permission to run applications on cs department server, members of eeStudent role have permission to run applications on ee department server, and members of student role have permission to run applications on a central university server. The student role is a junior role to both csStudent and eeStudent. The difference between π_{R1} and π_{R2} is that π_{R2} does not have the student role, and the permission to run applications on a central university server is assigned to both csStudent and eeStudent roles. π_{R2} has lower WSC than π_{R1}, but π_{R1} might be preferable for other reasons, for example, if rules that grant permissions on university servers are administered by the IT Department, and rules that grant permissions on a departmental server is administered by the owning department. Assuming suitable attribute data (a user attribute "dept" indicating the user's department, etc.), our algorithm applied to π_{R1} produces an ABAC policy π_{A1}, which has the same structure as π_{R1} and hence can be administered in the same way. In contrast, our algorithm applied to π_{R2} produces an ABAC policy π_{A2}, which has lower WSC than π_{A1} but cannot be administered in the same way as π_{A1}.

7. RELATED WORK

To the best of our knowledge, this paper presents the first algorithm specifically designed to mine ABAC policies from RBAC policies and attribute data, and the only prior work on mining ABAC policies is the Xu and Stoller's algorithm [4] that mines ABAC policies from ACLs and attribute data. The algorithm in [4] can be used to mine ABAC policies from RBAC policies and attribute data, by expanding RBAC policies into ACLs. However, that approach has significant disadvantages compared to the algorithm presented in this paper, mainly (1) the generated ABAC policy is less likely to have the desired structure, because the structure of the RBAC policy is not used to guide the structure of the ABAC policy, and (2) role membership information is not used to substitute for unavailable attribute information, leading to lower-level policies that use user identity instead of role membership information where the available attribute information is insufficient.

The next most closely related work is Xu and Stoller's algorithm for mining parameterized RBAC (PRBAC) policies from ACLs and attribute data [5]. Their PRBAC framework supports a simple form of ABAC, but quite limited compared to our ABAC framework. 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.

Less closely related work includes policy mining algorithms that take attribute data into account when mining RBAC policies (without parameters) from ACLs, e.g., [3, 6, 7].

8. ACKNOWLEDGEMENTS

This material is based upon work supported by ONR under Grant N00014-07-1-0928 and NSF under Grant CNS-0831298.

9. BIBLIOGRAPHY

[1] Ravi Sandhu and Edward Coyne. "Role-based access control models," IEEE Computer, pp. 38-47, 1996.

[2] Ravi Sandhu. "The authorization leap from rights to attributes: maturation or chaos?," In Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT), 2012.

[3] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin B. Calo, and Jorge Lobo. "Mining roles with multiple objectives," ACM Trans. Inf. Syst. Secur, vol. 13(4), 2010,

[4] Zhongyuan Xu and Scott D. Stoller. "Mining attribute-based access control policies," Submitted for publication. Available at http://arxiv.org/pdf/1306.2401.pdf, 2013.

[5] Zhongyuan Xu and Scott D. Stoller. "Mining parameterized role-based policies," In Proc. Third ACM Conference on Data and Application Security and Privacy (CODASPY), 2013.

[6] Alessandro Colantonio, Roberto Di Pietro, and Nino Vincenzo Verde. "A business-driven decomposition methodology for role mining," Computer & Security, vols. 31(7), pp. 844-855, 2012,

[7] Zhongyuan Xu and Scott D. Stoller. "Algorithms for mining meaningful roles," In Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 57-66, 2012.