Mining Relationship-Based Access Control Policies

Thang Bui
Stony Brook University
thang.bui@stonybrook.edu

Scott D. Stoller
Stony Brook University
stoller@cs.stonybrook.edu

Jiajie Li
Stony Brook University
jiajie.li@stonybrook.edu

ABSTRACT

Relationship-based access control (ReBAC) provides a high level of expressiveness and flexibility that promotes security and information sharing. We formulate ReBAC as an object-oriented extension of attribute-based access control (ABAC) in which relationships are expressed using fields that refer to other objects, and path expressions are used to follow chains of relationships between objects.

ReBAC policy mining algorithms have potential to significantly reduce the cost of migration from legacy access control systems to ReBAC, by partially automating the development of a ReBAC policy from an existing access control policy and attribute data. This paper presents an algorithm for mining ReBAC policies from access control lists (ACLs) and attribute data represented as an object model, and an evaluation of the algorithm on four sample policies and two large case studies. Our algorithm can be adapted to mine ReBAC policies from access logs and object models. It is the first algorithm for these problems.

ACM Reference format:
Thang Bui, Scott D. Stoller, and Jiajie Li. 2017. Mining Relationship-Based Access Control Policies. In *Proceedings of SACMAT '17, Indianapolis, IN, USA, June 21–23, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078878

1 INTRODUCTION

The term *relationship-based access control* (ReBAC) was introduced to describe access control policies expressed in terms of interpersonal relationships in social network systems (SNSs). The underlying principle of expressing access control policies in terms of chains of relationships between entities is equally applicable and beneficial in general computing systems: it increases expressiveness and often allows more natural policies. This paper presents ORAL (Object-oriented Relationship-based Access-control Language), a ReBAC language formulated as an object-oriented extension of ABAC. Relationships are expressed using attributes that refer to other objects, including subjects and resources, and path expressions are used to follow chains of relationships between objects. In ORAL, a ReBAC policy consists of a class model, an object model, and access control rules. Section 6 compares ORAL with previous ReBAC models.

This material is based on work supported in part by NSF Grants CNS-1421893, and CCF-1414078, ONR Grant N00014-15-1-2208, AFOSR Grant FA9550-14-1-0261, and DARPA Contract FA8650-15-C-7561. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

SACMAT '17, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/3078861.3078878

The cost of manually developing an initial high-level policy is a barrier to adoption of high-level policy models. *Policy mining* algorithms promise to drastically reduce this cost, by partially automating the process. There is a significant amount of research on role mining and some recent research on ABAC policy mining [11, 13–15]. There is no prior work on mining of ReBAC policies (or object-oriented ABAC policies with path expressions).

This paper defines the ReBAC policy mining problem and presents the first algorithm for mining ReBAC policies from ACLs and attribute data represented as object models. It is easy to show that the problem is NP-hard, based on the Xu and Stoller's proof that ABAC policy mining is NP-hard [15]. Since we desire an efficient and practical algorithm, our algorithm incorporates greedy heuristics and is not guaranteed to generate an optimal policy. Our algorithm has three phases. In the first phase, it iterates over tuples in the subject-permission relation, uses selected tuples as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover additional tuples in the subject-permission relation by replacing conditions on user attributes or resource attributes with constraints that relate user attributes with resource attributes. The algorithm greedily selects the highest-quality generalization according to a rule quality metric based primarily on the ratio of the number of previously uncovered subject-permission tuples covered by the rule to the rule's WSC. The first phase ends when the set of candidate rules covers the entire subject-permission relation. The second phase attempts to improve the policy by merging and simplifying candidate rules. The third phase selects the highest-quality candidate rules for inclusion in the mined policy. This high-level algorithm structure is based on Xu and Stoller's algorithm for mining ABAC policies from ACLs [15], but there are also many differences between the algorithms, as discussed in detail in Section 6.

Our algorithm can be adapted to mine ReBAC policies from access logs and object models, in a similar way as Xu and Stoller's algorithm [15] was adapted to mine ABAC policies from access logs and attribute data [13].

We evaluate our algorithm on four relatively small but non-trivial sample policies and on two much larger and more complex case studies developed by Decat, Bogaerts, Lagaisse, and Joosen based on the access control requirements for Software-as-a-Service (SaaS) applications offered real companies [7, 8]. We translate Decat *et al.*'s detailed natural-language descriptions of the policies into class models and ReBAC rules, omitting a few aspects left for future work, mainly temporal conditions, obligations, and policy administration. To the best of our knowledge, these two case studies are the largest rule-based policies (as measured by the number and complexity of the rules) on which any policy mining algorithm has been evaluated.

Our evaluation methodology is to start with a ReBAC policy, generate ACLs representing the subject-permission relation, run our algorithm, and compare the ReBAC policy mined from ACLs with the original ReBAC policy. For the four sample policies, the

mined policy is identical to the original policy, except for one minor syntactic variation in one conjunct of one condition of one rule of one sample policy (the variant is semantically equivalent to and equally simple as the original conjunct). For the e-document case study, our algorithm achieves roughly 80% to 90% similarity between the original and mined policies, depending on details of the comparison metric. For the workforce management case study, the mined policy is simpler than the original policy, and our algorithm achieves roughly 70% to 90% similarity between the original and mined policies, depending on the metric.

2 POLICY LANGUAGE

This section presents our policy language, ORAL. It contains common ABAC constructs, similar to those in [15], plus path expressions.

A *ReBAC policy* is a tuple $\pi = \langle CM, OM, Act, Rules \rangle$, where *CM* is a class model, *OM* is an object model, *Act* is a set of actions, and *Rules* is a set of rules.

A *class model* is a set of class declarations. A *class declaration* is a tuple $\langle className, parent, fields \rangle$ where *parent* is a class name or the empty string (indicating that the class does not have a parent), and *fields* is a set of field declarations. A *field declaration* is a tuple $\langle fieldName, type, multiplicity \rangle$, where *type* is a class name or Boolean, and *multiplicity* is optional, one, or many. The *multiplicity* specifies how many values of the specified type may be stored in the field and is "one" (also denoted "1", meaning exactly one), "optional" (also denoted "?", meaning zero or one), or "many" (also denoted "*", meaning any natural number). Boolean fields always have multiplicity 1. Every class implicitly contains a field "id" with type String. We keep the language minimal by not allowing user-defined fields with type string. However, their effect can be achieved using a field that refers to an object having the desired string as its id. Thus, the set of types in a policy contains Boolean, String, and the names of the declared classes. A *reference type* is any class name (used as a type).

An *object model* is a set of objects whose types are consistent with the class model and with unique values in the "id's . An *object* is a tuple $\langle className, fieldVals \rangle$, where *fieldVals* is a function that maps the names of fields of the specified class, including the id field and inherited fields, to values consistent with the types and multiplicities of the fields. The value of a field with multiplicity many is a set. The value of a field with multiplicity one or optional is a single value; the special placeholder $\perp$ is used when a field with multiplicity optional lacks an actual value. For an object $o = \langle c, fv \rangle$, let type($o$) = c and fVal(o) = fv.

A *condition* is a set, interpreted as a conjunction, of atomic conditions. We often refer to the atomic conditions as conjuncts. Informally, an atomic condition is a condition on the value of one field of one object. An *atomic condition* is a tuple $\langle p, op, val \rangle$, where p is a non-empty path, *op* is an operator, either "in" or "contains", and *val* is a constant value, either an atomic value or a set of atomic values. Note that *val* cannot equal or contain the placeholder $\perp$. A *path* is a sequence of field names. In examples, we usually write conditions with a logic-based syntax, for readability, using "$\in$" for "in" and "$\ni$" for "contains". For example, we may write $\langle$dept.id, in, {CompSci}$\rangle$ as dept.id $\in$ {CompSci}. We may use "=" as syntactic sugar for "in" when the constant is a singleton set; thus, the previous example may be written as dept.id=CompSci. Note that a condition may contain multiple atomic conditions on the same path.

A *constraint* is a set, interpreted as a conjunction, of atomic constraints. Informally, an atomic constraint expresses a relationship between the value of one field of one object (the subject issuing the request) and the value of one field of another object (the requested resource). An *atomic constraint* is a tuple $\langle p_1, op, p_2 \rangle$, where p_1 and p_2 are paths (possibly the empty sequence), and *op* is one of the following four operators: equal, in, contains, supseteq. The "contains" operator is the transpose of the "in" operator. Implicitly, the first path is relative to the requesting subject, and the second path is relative to the requested resource. The empty path represents the subject or resource itself. In examples, we usually write constraints with a logic-based syntax, for readability, using "=" for "equal" and "$\supseteq$" for "supseteq", and we prefix the subject path p_1 and resource path p_2 with "subject" and "resource", respectively. For example, $\langle$specialties, contains, topic$\rangle$ may be written as subject.specialties $\ni$ resource.topic. Other relational operators, such as $\subseteq$, could also be added; we omit them for now, since they are not needed for our case studies.

A *rule* is a tuple $\langle subjectType, subjectCondition, resourceType, resourceCondition, constraint, actions \rangle$, where *subjectType* and *resourceType* are class names, *subjectCondition* and *resourceCondition* are conditions, *constraint* is a constraint, *actions* is a set of actions, and the following well-formedness requirements are satisfied. Implicitly, the paths in *subjectCondition* and *resourceCondition* are relative to the requesting subject and requested resource, respectively. The *type of a path p* (relative to a specified class), denoted type(p), is the type of the last field in the path. The *multiplicity of a path p* (relative to a specified class), denoted multiplicity(p), is one if all fields on the path have multiplicity one, is many if any field on the path has multiplicity many, and is optional otherwise.

Well-formedness requirements on rules are as follows. (1) All paths are type-correct, assuming the subject and resource have type *subjectType* and *resourceType*, respectively. (2) (a) The two paths in the constraint have the same type, and (b) this type is not String. Part (a) reflects the assumption that comparing objects of different types is either meaningless or useless (since it would be equivalent to "false"). Part (b) prohibits constraints that compare identifiers of objects with different types, which would be meaningless. It does not reduce the expressiveness of the model, because a constraint violating it, such as specialties.id $\ni$ topic.id, can be written more simply as specialties $\ni$ topic. (3) The path in the condition does not have reference type. This reflects the fact that our language does not allow constants with reference type. (4) In conditions with operator "in", the path has multiplicity optional or one, and the constant is a set. This excludes sets of sets from the model. (5) In conditions with operator "contains", the path has multiplicity many, and the constant is an atomic value. (6) In constraints with operator "equal", both paths have multiplicity optional or one. (7) In constraints with operator "in", the first path has multiplicity optional or one, and the second path has multiplicity many. (8) In constraints with operator "contains", the first path has multiplicity many, and the second path has multiplicity optional or one. (9) In constraints with operator "supseteq", both paths have multiplicity many.

In examples, we prefix the path in the subject condition and resource condition with "subject" and "resource", respectively, for readability. For example, our project management policy contains the rule: A contractor working on a project can read and request to work on a non-proprietary task of the project whose required areas of expertise are among his/her areas of expertise. This is expressed as ⟨Contractor, true, Task, resource.isProprietary=false, subject.projects ∋ resource.project ∧ subject.expertise ⊇ resource.expertise, {read, request}⟩ .

For a rule $\rho = \langle st, sc, rt, rc, c, A\rangle$, let $\text{sType}(\rho) = st$, $\text{sCond}(\rho) = sc$, $\text{rType}(\rho) = rt$, $\text{rCond}(\rho) = rc$, $\text{con}(\rho) = c$, and $\text{acts}(\rho) = A$.

An *subject-permission* tuple is a tuple $\langle s, r, a\rangle$, where s and r are objects, and a is an action. This tuple means that subject s is permitted to perform action a on resource r. A *subject-permission relation* is a set of such tuples.

Given an class model, object model, object o, and path p, let $\text{nav}(o, p)$ be the result of navigating (a.k.a. following or dereferencing) path p starting from object o. The class model and object model are implicit arguments to this relation and the following relations. We elide these arguments, because in our setting, they are unchanging in the context of a given policy, so making them explicit arguments would just add clutter. The result might be no value, represented by $\perp$, an atomic value, or a set. A set may be obtained if any field along the path (not necessarily the last field) has multiplicity many. This is like the semantics of path navigation in UML's Object Constraint Language (OCL) (http://www.omg.org/spec/OCL/).

An object o *satisfies* an atomic condition $c = \langle p, op, val\rangle$, denoted $o \models c$, if $(op = \text{in} \wedge \text{nav}(o, p) \in val) \vee (op = \text{contains} \wedge \text{nav}(o, p) \ni val)$. The *meaning* of a condition c relative to a class C, denoted $[\![c]\!]_C$ is the set of instances of C (in the implicitly given object model) that satisfy c. A condition c *characterizes* a set O of objects of class C if O is the meaning of c relative to C.

Objects o_1 and o_2 *satisfy* an atomic constraint $c = \langle p_1, op, p_2\rangle$, denoted $\langle o_1, o_2\rangle \models c$, if $(op = \text{equal} \wedge \text{nav}(o_1, p_1) = \text{nav}(o_2, p_2)) \vee (op = \text{in} \wedge \text{nav}(o_1, p_1) \in \text{nav}(o_2, p_2)) \vee (op = \text{contains} \wedge \text{nav}(o_1, p_1) \ni \text{nav}(o_2, p_2)) \vee (op = \text{supseteq} \wedge \text{nav}(o_1, p_1) \supseteq \text{nav}(o_2, p_2))$.

A subject-permission tuple $\langle s, r, a\rangle$ *satisfies* a rule $\rho = \langle st, sc, rt, rc, c, A\rangle$, denoted $\langle s, r, o\rangle \models \rho$, if $\text{type}(s) = st \wedge s \models sc \wedge \text{type}(r) = rt \wedge r \models rc \wedge \langle s, r\rangle \models c \wedge a \in A$.

The *meaning* of a rule ρ, denoted $[\![\rho]\!]$, is the subject-permission relation it induces, defined as $[\![\rho]\!] = \{\langle s, r, a\rangle \in OM \times OM \times Act \mid \langle s, r, a\rangle \models \rho\}$.

The *meaning* of a ReBAC policy π, denoted $[\![\pi]\!]$, is the subject-permission relation it induces, defined as the union of the meanings of its rules.

3 PROBLEM DEFINITION

An *access control list (ACL) policy* is a tuple $\langle CM, OM, Act, SP_0\rangle$, where CM is a class model, OM is an object model, Act is a set of actions, and $SP_0 \subseteq OM \times OM \times Act$ is a subject-permission relation. Conceptually, SP_0 is the union of the resources' access control lists.

An ReBAC policy π is *consistent* with an ACL policy $\langle CM, OM, Act, SP_0\rangle$ if they have the same class model, object model, and actions and $[\![\pi]\!] = SP_0$.

An ReBAC policy consistent with a given ACL policy can be trivially constructed, by creating a separate rule corresponding to each subject-permission tuple in the ACL policy, using a condition "id=..." to identify the relevant subject and resource. Of course, such a ReBAC policy is as verbose and hard to manage as the original ACL policy. Therefore, we must decide: among ReBAC policies consistent with a given ACL policy π_0, which ones are preferable? We adopt two criteria.

One criterion is that the "id' field should be avoided when possible, because policies that use this field are (to that extent) identity-based, not attribute-based or relationship-based. Therefore, our definition of ReBAC policy mining requires that these attributes are used only when necessary, i.e., only when every ReBAC policy consistent with π_0 contains rules that use them.

The other criterion is to maximize a policy quality metric. A *policy quality metric* is a function Q_{pol} from ReBAC 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 *ReBAC policy mining problem* is: given an ACL policy $\pi_0 = \langle CM, OM, Act, SP_0\rangle$ and a policy quality metric Q_{pol}, find a set *Rules* of rules such that the ReBAC policy $\pi = \langle CM, OM, Act, Rules\rangle$ is consistent with π_0, uses the "id" field only when necessary, and has the best quality, according to Q_{pol}, among such policies.

The policy quality metric that our algorithm aims to optimize is *weighted structural complexity* (WSC), a generalization of policy size first introduced for RBAC policies [12] and later extended to ABAC [15]. Minimizing policy size is consistent with prior work on ABAC mining and role mining and with usability studies showing that more concise access control policies are more manageable [1]. Informally, the WSC of a ReBAC policy is a weighted sum of the numbers of elements of each kind in the policy. Formally, the WSC of a ReBAC policy π, denoted $\text{WSC}(\pi)$, is the sum of the WSC of its rules, defined bottom-up as follows. The WSC of an atomic condition $\langle p, op, val\rangle$ is $|p| + |val|$, where $|p|$ is the length of path p, and $|val|$ is 1 if val is an atomic value and is the cardinality of val if val is a set. The WSC of an atomic constraint $\langle p_1, op, p_2\rangle$ is $|p_1|+|p_2|$. The WSC of a condition c, denoted $\text{WSC}_{\text{cndn}}(c)$, is the sum of the WSC of the constituent atomic conditions. The WSC of a constraint c, denoted $\text{WSC}_{\text{cnst}}(c)$, is the sum of the WSC of the constituent atomic constraints. The WSC of a rule is $\text{WSC}(\langle st, sc, rt, rc, c, A\rangle) = w_1\text{WSC}_{\text{cndn}}(sc) + w_1\text{WSC}_{\text{cndn}}(rc) + w_2\text{WSC}_{\text{cnst}}(c) + w_3|A|$, where $|A|$ is the cardinality of set A, and the w_i are user-specified weights.

4 ALGORITHM

This section presents our algorithm. It is based on the ABAC policy mining algorithm in [15]. The main differences are summarized in Section 6.

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

```
// Phase 1: Create a set Rules of candidate rules that covers SP0.
Rules = ∅
// uncovSP contains tuples in SP0 that are not covered by Rules
uncovSP = SP0.copy()
while ¬uncovSP.isEmpty()
  // Select an uncovered tuple as a "seed".
  ⟨s, r, a⟩ = some tuple in uncovSP
  cc = candidateConstraint(s, r)
  // ss contains subjects with permission ⟨r, a⟩ and that have
  // the same candidate constraint for r as s
  ss = {s′ ∈ OM | type(s′) = type(s) ∧ ⟨s′, r, a⟩ ∈ SP0
                  ∧ candidateConstraint(s′, r) = cc}
  addCandidateRule(type(s), ss, type(r), {r}, cc, {a}, uncovSP, Rules)
  // sa is set of actions that s can perform on r
  sa = {a′ ∈ Act | ⟨s, r, a′⟩ ∈ SP0}
  addCandidateRule(type(s), {s}, type(r), {r}, cc, sa, uncovSP, Rules)
end while
// Phase 2: Combine rules using least upper bound and inheritance.
// Also, simplify them and remove redundant rules.
mergeRulesLUBandSimplify(Rules)
mergeRulesInheritance(Rules)
mergeRulesLUBandSimplify(Rules)
// Remove redundant rules
while Rules contains rules ρ and ρ′ such that [[ρ]] ⊆ [[ρ′]]
  Rules.remove(ρ)
end while
// Phase 3: Select high quality rules into Rules′.
Rules′ = ∅
Repeatedly move highest-quality rule from Rules to Rules′ until
  ∑ρ∈Rules′ [[ρ]] ⊇ SP0, using SP0 \ [[Rules′]] as second argument
  to Qrul, and discarding a rule if it does not cover any tuples in
  SP0 currently uncovered by Rules′.
return Rules′

// Repeatedly merge rules using least upper bound and
// simplify them, until this has no effect
function mergeRulesLUBandSimplify(Rules)
mergeRulesLUB(Rules)
while simplifyRules(Rules) && mergeRulesLUB(Rules)
  skip
end while
```

Figure 1: Policy mining algorithm.

The workset *uncovSP* in Figure 1 is a priority queue sorted in descending lexicographic order by the quality Q_{sp} of the subject-permission tuple. Informally, $Q_{\text{sp}}(\langle s, r, a\rangle)$ is a triple whose first two components are the frequency of permission $\langle r, a\rangle$ and subject s, respectively, i.e., their numbers of occurrences in SP_0, and whose third component (included as a tie-breaker to ensure a total order) is the string representation of the tuple.

$$\begin{aligned} \text{freq}(\langle r, a\rangle) &= |\{\langle s', r', a'\rangle \in SP_0 \mid r' = r \wedge a' = a\}| \\ \text{freq}(s) &= |\{\langle s', r', a'\rangle \in SP_0 \mid s' = s\}| \\ Q_{\text{sp}}(\langle s, r, a\rangle) &= \langle \text{freq}(\langle r, a\rangle), \text{freq}(s), \text{toString}(\langle \text{s, r, a}\rangle)\rangle \end{aligned}$$

```
function candidateConstraint(s, r)
// cc is the set of type-correct candidate constraints
cc = ∅
for T in (reach(type(s)) ∩ reach(type(r)))
  // add candidate constraints where the paths have type T
  for p1 in paths(type(s), T)
    for p2 in paths(type(r), T)
      cc.add(⟨p1, opFromMul(multiplicity(p1), multiplicity(p2)), p2⟩)
    end for
  end for
end for
return {c ∈ cc | ⟨s, r⟩ ⊨ c}
```

Figure 2: Compute candidate constraints for subject s and resource r

The function candidateConstraint(s, r) in Figure 2 returns a set containing all the atomic constraints that hold between resource r and subject s. It first computes a set cc of candidate constraints using type-correct shortest paths to each type T reachable from both type(s) and type(r) in the graph graph(CM), which has a vertex for each class, and an edge from c_1 to c_2 if c_1 has a field with type c_2. It then selects and returns the candidate constraints satisfied by $\langle s, r\rangle$. This algorithm infers only constraints where the paths have reference types. It could easily be extended to infer constraints where the paths have type Boolean, but such constraints do not arise in our current case studies. It uses the following auxiliary functions. reach(T) returns the set of classes reachable from T in graph(CM), including their superclasses. **function** paths(T, T') returns the set of shortest paths from T to T' in graph(CM). **function** opFromMul(m, m') returns the relational operator suitable for left and right operands with multiplicity m and m', respectively, defined by the following case statement on $\langle m, m'\rangle$: $\langle$many, many$\rangle \Rightarrow$ supseteq | $\langle$many, _$\rangle \Rightarrow$ contains | $\langle$_, many$\rangle \Rightarrow$ in | $\langle$_, _$\rangle \Rightarrow$ equal.

We extend this function to also produce non-shortest paths. These extensions are not reflected in the pseudo-code. Specifically, we extend paths(T, T') to return paths with length at most dist(T, T') + SPED and dist(T, T') + RPED when T = type(s) and T = type(r), respectively, where dist(T, T') is the length of shortest paths from T to T' in graph(CM), and SPED (mnemonic for "subject path extra distance") and RPED (mnemonic for "resource path extra distance") are parameters of the algorithm. In order to limit the overall complexity of a candidate constraint, we also introduce a parameter MTPL (mnemonic for "maximum total path length") that limits the sum of the path lengths in a constraint; specifically, in the inner loop, a candidate constraint is constructed only if $|p_1| + |p_2| \leq$ MTPL.

The function addCandidateRule$(st, s_s, rt, s_r, cc, s_a, uncovSP, Rules)$ in Figure 3 calls computeCondition to compute conditions sc and rc that characterizes s_s and s_r, respectively. MSPL and MRPL are the maximum path length for paths in the subject condition and resource condition, respectively; they are parameters of the algorithm. addCandidateRule then constructs a rule $\rho = \langle st, sc, rt, rc, \emptyset, s_a\rangle$, calls generalizeRule to generalize ρ to ρ' and adds ρ' to candidate

function addCandidateRule($st, s_s, rt, s_r, cc, s_a, uncovSP, Rules$)
// *Construct a rule ρ that covers subject-permission tuples*
// $\{\langle s, r, a\rangle \mid s \in s_s \wedge r \in s_r \wedge a \in s_a\}$.
sc = computeCondition(s_s, st, MSPL);
rc = computeCondition(s_r, rt, MRPL)
$\rho = \langle st, sc, rt, rc, \emptyset, s_a\rangle$
ρ' = generalizeRule(ρ, cc, $uncovSP$, $Rules$)
$Rules$.add(ρ')
$uncovSP$.removeAll($[\![\rho']\!]$)

Figure 3: Compute a candidate rule and add it to *Rules*

rule set *Rules*. The details of the functions called by addCandidateRule are described next.

The function computeCondition(O, C, L) computes a condition C that characterizes the set O of objects of type C using paths of length at most L. A path with multiplicity optional or one appears in at most one conjunct, of the form $\langle p, \text{in}, V\rangle$ where V is the collected values of $o.p$ for o in O. A path with multiplicity may appear in multiple conjuncts, of the form $\langle p, \text{contains}, val\rangle$ where val is in the intersection of the values of $o.p$ for o in O. First, paths not containing the id field are considered. If the resulting condition does not characterize O, then (by construction) it is an over-approximation, and a conjunct using the "id" field is added to ensure that the resulting condition characterizes O. The condition returned by computeCondition might not be minimum-sized among conditions that characterize O: possibly some conjuncts can be deleted without changing the condition's meaning. We defer minimization of the condition until after the call to generalizeRule (described below), because minimizing the condition before that would reduce opportunities to find constraints in generalizeRule.

A rule ρ' is *valid* if $[\![\rho']\!] \subseteq SP_0$.

The function generalizeRule(ρ, cc, $uncovSP$, $Rules$) attempts to generalize rule ρ by adding some of the atomic constraints in cc to ρ and eliminating the conjuncts of the subject condition and resource condition that use the same paths as those constraints. A rule obtained in this way is called a *generalization* of ρ. It is more general in the sense that it refers to relationships instead of specific values. The meaning of a generalization of ρ is a superset of the meaning of ρ. In more detail, generalizeRule sorts cc in the order described below, tries to add the constraints in every subsequence of cc to ρ, and if any of the resulting generalized rules is valid, it returns the highest-quality rule among them according to the rule quality metric described below, otherwise it returns ρ. When trying to add a constraint c in cc to a rule ρ, generalizeRule first tries removing the conjuncts of the subject condition and resource condition that use the same paths as c. If the resulting rule is invalid, it attempts a more conservative generalization by removing only the conjunct in the subject condition that uses the same path as c. If that rule is also invalid, it instead removes only the conjunct in the resource condition that uses the same path as c. If that rule is also valid, then there is no valid generalization of ρ using c.

generalizeRule sorts cc because the order in which candidate constraints are considered can affect the resulting generalized rule. For example, suppose adding candidate constraint c_1 keeps the rule valid and removes a conjunct in the subject condition, and that adding candidate constraint c_2 removes a conjunct in the subject condition and a conjunct in the resource condition, and yields a higher-quality resulting rule if the resulting rule is valid. Suppose, further, that adding c_2 keeps the rule valid only if c_1 has already been added. In this case, the highest-quality generalization will be created only if c_1 is considered before c_2. To sort cc, we temporarily add each constraint c in cc to ρ (in the same way as described above) and, if this leads to a valid generalization of ρ, compute the number of subject-permission tuples in $uncovSP$ covered by the resulting rule, and then sort cc in descending order by these values. If adding c does not lead to a valid generalization of ρ, c is useless and can be removed from cc.

A *rule quality metric* is a function $Q_{\text{rul}}(\rho, SP)$ that maps a rule ρ to a totally-ordered set, with the order chosen such that larger values indicate higher quality. The second argument SP is a set of subject-permission tuples. Based on our primary goal of minimizing the mined policy's WSC, a secondary preference for rules with more constraints, and a tertiary preference for rules with shorter paths in constraints, we define

$$Q_{\text{rul}}(\rho, SP) = \langle |\, [\![\rho]\!] \cap SP|/\text{WSC}(\rho),\ |\text{con}(\rho)|,\ 1/\text{TCPL}(\rho)\rangle$$

where TCPL(ρ) ("total constraint path length") is the sum of the lengths of the paths used in the constraints of ρ.

The preference for more constraints is a heuristic, based on the observation that rules with more constraints tend to be more general than other rules with the same $|\, [\![\rho]\!] \cap SP|/\text{WSC}(\rho)$ (such rules typically have more conjuncts) and hence lead to lower WSC for the policy. In generalizeRule, $uncovSP$ is the second argument to Q_{rul}, so $[\![\rho]\!] \cap SP$ is the set of subject-permission tuples in SP_0 that are covered by ρ and not covered by existing rules.

The function mergeRulesLUB($Rules$) attempts to improve the quality of $Rules$ by merging pairs of rules that have the same subject type, resource type, and constraint by taking the least upper bound of their subject conditions, the least upper bound of their resource conditions, and the union of their sets of actions. The *least upper bound* of conditions c_1 and c_2, denoted $c_1 \sqcup c_2$, is

$$\begin{aligned}&\{\langle p, \text{in}, val\rangle \mid (\exists val_1, val_2.\ \langle p, \text{in}, val_1\rangle \in c_1 \wedge \langle p, \text{in}, val_2\rangle \in c_2\\ &\qquad\qquad \wedge\ val = val_1 \cup val_2)\}\\ &\cup \{\langle p, \text{contains}, val\rangle \mid \langle p, \text{contains}, val\rangle \in c_1\\ &\qquad\qquad \wedge\ \langle p, \text{contains}, val\rangle \in c_2)\}.\end{aligned}$$

Note that the meaning of the merged rule ρ_{mrg} is a superset of the meanings of the rules ρ_1 and ρ_2 being merged. If the merged rule ρ_{mrg} is valid, then it replaces ρ_1 and ρ_2 in $Rules$. mergeRulesLUB($Rules$) updates its argument $Rules$ in place, and it returns a Boolean indicating whether any rules were merged.

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

(1) It eliminates conjuncts from the subject and resource conditions when this preserves validity. Since removing one conjunct might prevent removal of another conjunct, it searches for a set of conjuncts that maximizes the quality of the resulting rule. To limit the cost, we introduce a parameter MCSE (mnemonic

for "maximum conjuncts to simplify exhaustively"). If the number of conjuncts is at most MCSE, the algorithm tries removing every subset of conjuncts. If the number of conjuncts exceeds MCSE, the algorithm sorts the conjuncts in descending lexicographic order by Q_{ac} (quality metric for atomic conditions) and then attempts to remove them linearly in the sorted order, where $Q_{\text{ac}}(\langle p, op, val\rangle) = \langle |val|, |p|, \text{isId}(p), \text{toString(p)}\rangle$, where $|val|$ is 1 if *val* is an atomic value and is the cardinality if *val* is a set, and $\text{isId}(p)$ is 1 if p is "id" and is 0 otherwise. The last component of Q_{ac} is included as a tie-breaker to ensure a total order. (2) It eliminates atomic constraints when this preserves validity. It searches for the set of atomic constraints to remove that maximizes the quality of the resulting rule, while preserving validity. (3) It eliminates overlapping actions between rules. Specifically, an action a in a rule ρ is removed if there is another rule ρ' in the policy such that $\text{sCond}(\rho') \subseteq \text{sCond}(\rho) \wedge \text{rCond}(\rho') \subseteq \text{rCond}(\rho) \wedge \text{con}(\rho') \subseteq \text{con}(\rho) \wedge a \in \text{acts}(\rho')$. (4) It eliminates actions when this preserves the meaning of the policy. In other words, it removes an action a in rule ρ if all the subject-permission tuple covered by a in ρ are covered by other rules in the policy. Note that (3) is a special case of (4), listed separately to ensure that this special case takes precedence.

The function mergeRulesInheritance(*Rules*) attempts to merge a set of rules if their subject types or resource types have a common superclass and all the other components of the rule are the same. In this case, it replaces that set of rules with a single rule whose subject type or resource type is the most general superclass for which the merged rule is valid. For example, rules $\langle st_1, sc, rt, rc, c, A\rangle$ and $\langle st_2, sc, rt, rc, c, A\rangle$ are replaced with $\rho_{\text{mrg}} = \langle st', sc, rt, rc, c, A\rangle$ if ρ_{mrg} is valid, and st' is a superclass of st_1 and st_2, and these conditions do not hold for any superclass of st'.

4.1 Sample Policies and Case Studies

We developed six ReBAC policies: four sample policies, which have non-trivial and realistic rules, but are smaller and not directly based on the policy of a particular organization, and two large case studies, based on the policies of real (but anonymous) companies, as described by Decat *et al.* [7, 8]. Each policy has handwritten class model and rules, and a pseudorandom synthetic object model generated by a policy-specific algorithm. Each object model generation algorithm is parameterized by a size parameter N.

The *Electronic Medical Record (EMR) sample policy*, based on the EBAC policy in [2], controls access by physicians and patients to electronic medical records. The numbers of physicians, patients, medical records, and hospitals are proportional to N. A sample rule is "A physician at a facility can view a medical record for a consultation with any physician at that facility by a patient still registered at the facility", expressed as ⟨ Physician, true, MedicalRecord, true, subject.affiliation = resource.consultation.physician.affiliation ∧ subject.affiliation ∈ resource.consultation.patient.registrations, {view}⟩.

The *healthcare sample policy*, based on the ABAC policy in [15], 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). The numbers of wards, teams, doctors, nurses, teams, patients, and agents are proportional to N. A sample rule is "A doctor 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", expressed as ⟨ Doctor, true, HealthRecordItem, true, subject.teams contains resource.record.patient.treatingTeam ∧ subject.specialties ⊇ resource.topics, {read}⟩, where HealthRecordItem.record is the health record containing the HR item.

The *project management sample policy*, based on the ABAC policy in [15], controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. The numbers of departments, projects, tasks, and users of each type are proportional to N. A sample rule appears in Section 2.

The *university sample policy*, based on the ABAC policy in [15], controls access by students, instructors, teaching assistants (TAs), department chairs, and staff in the registrar's office and admissions office to applications (for admission), gradebooks, transcripts, and course schedules. The numbers of departments, students, faculty, and applicants for admission are proportional to N.

The *e-document case study*, based on [7], is for a SaaS multi-tenant e-document processing application. The application allows tenants to distribute documents to their customers, either digitally or physically (by printing them and employing postal mail). The overall policy contains rules governing document access and administrative operations by employees of the e-document company, such as helpdesk operators and application administrators. It also contains specific policies for some sample tenants. The numbers of employees of each tenant, registered users of each customer organization, and documents are proportional to N.

The *workforce management case study*, based on [8], is for a SaaS workforce management application provided by a company called eWorkforce which handles the workflow planning and supply management for product or service appointments (e.g., install or repair jobs). Tenants (i.e., eWorkforce customers) can create tasks on behalf of their customers. Technicians working for eWorkforce, one of its workforce suppliers, or one of the subcontractors of one of the workforce supplies receive work orders to work on those tasks, and an appointment is scheduled if appropriate. The numbers of helpdesk suppliers, workforce providers, sub-contractors, helpdesk operators, contracts, work orders, etc., are proportional to N.

The algorithm parameters are set as follows in our experiments. For all policies, MCSE = 5. For EMR, MSPL = 3, MRPL = 4, SPED = 0, RPED = 1, and MTPL = 4. For healthcare, project management, and university, MSPL = 3, MRPL = 3, SPED = 0, RPED = 0, and MTPL = 4. For e-document, MSPL = 4, MRPL = 4, SPED = 0, RPED = 0, and MTPL = 4. For workforce management, MSPL = 3, MRPL = 3, SPED = 0, RPED = 2, and MTPL = 5.

5 EVALUATION

To evaluate the effectiveness of our algorithm, we start with a ReBAC policy, generate ACLs representing the subject-permission relation, run our algorithm on the ACLs along with the class model and object model, and compare the mined ReBAC policy with the original ReBAC policy. If the mined policy is similar to the original policy, the algorithm succeeded in discovering the rules that are implicit in the ACLs.

We compare the mined policy with the original policy and with a simplified version of the original policy, obtained by applying simplifyRules. When the algorithm fails to produce high-level rules, the mined policy differs from both the original and simplified original. When it produces high-level rules that have similar or lower WSC than the original handwritten rules, but express some aspects in a different high-level way, the mined policy differs from the original but agrees with the simplified original. Thus, comparison with the simplified original policy is a more robust measure of the algorithm's ability to discover high-level rules.

5.1 Policy Similarity Metrics

Both of our policy similarity metrics are normalized to range from 0 (completely different) to 1 (identical).

Syntactic Similarity. Syntactic similarity measures the fraction of atomic conditions, atomic constraints, and actions that rules or policies have in common. The Jaccard similarity of sets is $J(S_1, S_2) = |S_1 \cap S_2| \,/\, |S_1 \cup S_2|$. The *syntactic similarity of rules* $\rho_1 = \langle st_1, sc_1, rt_1, rc_1, c_1, A_1\rangle$ and $\rho_2 = \langle st_2, sc_2, rt_2, rc_2, c_2, A_2\rangle$ is 0 if $st_1 \neq st_2 \vee rt_1 \neq rt_2$ and is the average of $J(sc_1, sc_2)$, $J(rc_1, rc_2)$, $J(c_1, c_2)$ and $J(A_1, A_2)$ otherwise. The *syntactic similarity of rule sets* $Rules_1$ and $Rules_2$ is the average, over rules ρ in $Rules_1$, of the syntactic similarity between ρ and the most similar rule in $Rules_2$. The *syntactic similarity* of policies π_1 and π_2 is the maximum of the syntactic similarities of the sets of rules in the policies, considered in both orders.

Semantic Similarity. Semantic similarity measures the fraction of granted entitlements that rules or policies have in common. The *semantic similarity of rules* ρ_1 and ρ_2 is $J([\![\rho_1]\!], [\![\rho_2]\!])$. We extend this to *per-rule semantic similarity of policies* in exactly the same way that syntactic similarity of rules is extended to syntactic similarity of policies. Note that this metric measures similarity of the meanings of the rules in the policies, not similarity of the overall meanings of the policies.

5.2 Policy Similarity Results

Figure 4 shows the sizes of the policies and the results of policy similarity measurements. Each data point is the average over 30 pseudo-random object models. We set all weights w_i in the definition of WSC to 1.

For the *healthcare policy*, *project management policy*, and *university policy*, the original, simplified original, and mined policies are identical. For the *EMR policy*, the original and simplified original policies are identical, and the mined policy has perfect per-rule semantic similarity with them and nearly perfect (0.98) average syntactic similarity with them.

The *e-document case study* is the most difficult for our algorithm. The algorithm does well on 37 of the 39 input rules, achieving an average syntactic similarity of 0.90 and an average semantic similarity of 0.92 with the simplified original policy. The mined policy's WSC is significantly higher than the WSC of the original policy, mostly due to the algorithm's difficulty with two rules.

For the *workforce management case study*, the algorithm achieves an average syntactic similarity of 0.74 and an average semantic similarity of 0.93 with the simplified original policy. The mined policy's WSC is lower than the WSC of the original policy, although higher than the WSC of the simplified original policy.

5.3 Performance Results

Figure 5 shows the running time as a function of ACL policy size $|SP_0|$ on an Intel i7-6700HQ CPU for both case studies and some sample policies. Each data point is the average over 10 pseudo-random object models. Error bars show 95% confidence intervals using Student's t-distribution. The running times on these policies are low-order polynomials in $|SP_0|$: the slopes of the best-fit lines on a log-log plot of the data are 1.4 for workforce management, 1.5 for e-document, 2.7 for project management, and 3.2 for healthcare.

The results for the two case studies are encouraging indicators of the algorithm's scalability: the algorithm can mine dozens of complex rules from ACLs with several thousand entries in several minutes, and the running time grows roughly proportional to $|SP_0| \times \sqrt{|SP_0|}$.

6 RELATED WORK

6.1 Policy Models

Entity-Based Access Control (EBAC) [2] is the policy model most closely related to ours. EBAC is quite similar to ORAL, except that it is based on entity-relationship models, instead of object-oriented models, and hence lacks the concept of inheritance, which ORAL includes. EBAC's expression language includes quantifiers, and ORAL does not, although some conditions that require quantifiers in their language can be expressed in ORAL using the built-in binary relations on sets, such as $\supseteq$.

Several ReBAC models have been proposed, by Carminati, Ferrari, and Perego [4], Fong [9], Cheng, Park, and Sandhu [5], Hu, Ahn, and Jorgensen [10], Crampton and Sellwood [6], and others. Some are designed specifically for OSNs, while others are designed for general use. Our model differs from all of them because it is designed as a (nearly) minimal extension of a typical ABAC language, and the extension is achieved by adopting an object-oriented model and incorporating standard object-oriented concepts, notably path expressions, like in UML's Object Constraint Language (OCL) (http://www.omg.org/spec/OCL/). None of these ReBAC models are based on general object-oriented data models. None of these ReBAC models can express constraints between fields (a.k.a. attributes) of different entities, such as the constraint "subject.expertise $\supseteq$ resource.expertise" in the sample rule in Section 2. In this regard, ORAL is significantly more expressive.

On the other hand, ORAL lacks some features found in these ReBAC models. For example, all of the languages cited above include some form of transitive closure, and ORAL does not. The languages in [2, 3, 5, 9] include some form of negation, and ORAL does not, although some conditions expressed with negation in other frameworks can be expressed in ORAL using atomic conditions of the form $\langle p, \text{in}, \emptyset\rangle$. The modal-logic-based policy languages in [3, 9] include formulas that specify graph patterns, not merely paths. Many realistic applications do not require these language features, but they are useful for some applications. These features can easily be added to our policy language. However, developing policy mining algorithms that fully exploit them may be difficult. We leave that challenge for future work.

Policy	#rules	N	#obj	#field	$\|SP_0\|$	WSC			Mined vs Orig		Mined vs. SimpOrig	
						Orig.	SimpOrig	Mined	SynSim	SemSim	SynSim	SemSim
EMR	6	15	344	854	708	49	49	49	0.98	1	0.98	1
healthcare	9	5	737	1806	2207	54	54	54	1	1	1	1
project mgmt.	13	5	181	300	322	76	76	76	1	1	1	1
university	10	5	731	908	2439	54	54	54	1	1	1	1
e-document	39	125	421	2045	2687	359	250	463	0.85	0.79	0.90	0.92
workforce mgmt.	27	10	411	1123	1739	262	208	223	0.68	0.92	0.74	0.93

Figure 4: Policy sizes and policy similarities. For the given value of N, #obj is the average number of objects in the object model, #field is the average sum of the number of instances of each class times the number of fields in that class, SynSim is syntactic similarity, and SemSim is per-rule semantic similarity.

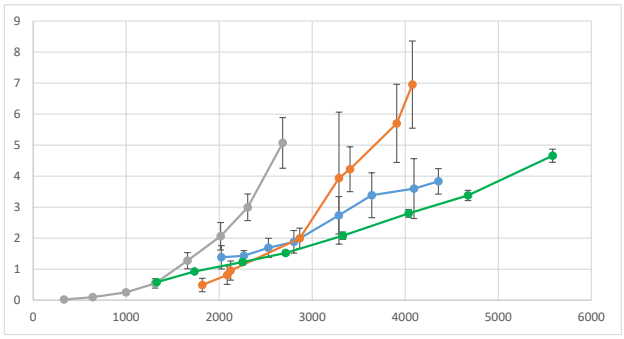

Figure 5: Running time, in minutes, as a function of $|SP_0|$ for project management (gray), healthcare (orange), e-document (blue), and workforce management (green).

The languages in [3, 5, 6, 9] allow every relation to be traversed in reverse. ORAL, like EBAC and OCL, does not; instead, the policy designer explicitly enables reverse traversal where appropriate by including a field in the reverse direction (this corresponds to using a bidirectional association in the UML class model).

6.2 Policy Mining

The most closely related prior work on policy mining is for ABAC policies without path expressions. Xu and Stoller developed the first algorithms for mining ABAC policies, from attribute data plus ACLs [15], roles [14], or access logs [13]. Our algorithm is based on their algorithm for mining ABAC policies from ACLs [15]. Adapting their algorithm to be suitable for ReBAC mining required many changes, most notably generalization of loops over attributes to iterate over paths when generating conditions and constraints; specifically, we introduce the idea of generating constraints based on shortest paths and nearly-shortest paths between classes in the graph representation of the class model. The technique for merging rules for sibling classes into a rule for an ancestor class is also new. We also modified the algorithm to accommodate changes in the supported relational operators: in conditions, we allow "in" and "contains", instead of "equal" and "supseteq" in [15]; in constraints, we allow "in" in addition to "equal", "contains", and "supseteq" allowed in [15]. We also introduced several techniques to limit and prioritize the paths being considered.

REFERENCES

[1] Matthias Beckerle and Leonardo A. Martucci. 2013. 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, Article 2, 11 pages.

[2] Jasper Bogaerts, Maarten Decat, Bert Lagaisse, and Wouter Joosen. 2015. Entity-Based Access Control: supporting more expressive access control policies. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, 291–300. https://lirias.kuleuven.be/handle/123456789/521795

[3] Glenn Bruns, Michael Huth, Philip Fong, and Ida Siahaan. 2012. Relationship-Based Access Control: Its Expression and Enforcement Through Hybrid Logic. In *Proc. Second ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 117–124.

[4] Barbara Carminati, Elena Ferrari, and Andrea Perego. 2009. Enforcing access control in Web-based social networks. *ACM Transactions on Information and System Security* 13, 1 (2009), 1–38.

[5] Yuan Cheng, Jaehong Park, and Ravi S. Sandhu. 2012. A User-to-User Relationship-Based Access Control Model for Online Social Networks. In *Proc. 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec) (Lecture Notes in Computer Science)*, Vol. 7371. Springer, 8–24.

[6] Jason Crampton and James Sellwood. 2014. Path conditions and principal matching: a new approach to access control. In *Proc. 19th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 187–198.

[7] Maarten Decat, Jasper Bogaerts, Bert Lagaisse, and Wouter Joosen. 2014. *The e-document case study: functional analysis and access control requirements.* CW Reports CW654. Department of Computer Science, KU Leuven. https://lirias.kuleuven.be/handle/123456789/440202

[8] Maarten Decat, Jasper Bogaerts, Bert Lagaisse, and Wouter Joosen. 2014. *The workforce management case study: functional analysis and access control requirements.* CW Reports CW655. Department of Computer Science, KU Leuven. https://lirias.kuleuven.be/handle/123456789/440203

[9] Philip W. L. Fong. 2011. Relationship-based access control: protection model and policy language. In *Proc. First ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 191–202.

[10] Hongxin Hu, Gail-Joon Ahn, and Jan Jorgensen. 2013. Multiparty access control for online social networks: model and mechanisms. *IEEE Transactions on Knowledge and Data Engineering* 25, 7 (2013), 1614–1627.

[11] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. 2015. Evolutionary Inference of Attribute-based Access Control Policies. In *Proceedings of the 8th International Conference on Evolutionary Multi-Criterion Optimization (EMO): Part I (Lecture Notes in Computer Science)*, Vol. 9018. Springer, 351–365.

[12] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin B. Calo, and Jorge Lobo. 2010. Mining Roles with Multiple Objectives. *ACM Trans. Inf. Syst. Secur.* 13, 4, Article 36 (2010), 36:1–36:35 pages.

[13] Zhongyuan Xu and Scott D. Stoller. 2014. Mining Attribute-Based Access Control Policies from Logs. In *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2014) (Lecture Notes in Computer Science)*, Vijay Atluri and Guenther Pernul (Eds.), Vol. 8566. Springer-Verlag, 276–291.

[14] Zhongyuan Xu and Scott D. Stoller. 2014. Mining Attribute-Based Access Control Policies from Role-Based Policies. In *Proceedings of the 10th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT 2013)*. IEEE Press.

[15] Zhongyuan Xu and Scott D. Stoller. 2015. Mining Attribute-based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing* 12, 5 (September–October 2015), 533–545.