Mining Relationship-Based Access Control Policies from Incomplete and Noisy Data[⋆]

Thang Bui, Scott D. Stoller, and Jiajie Li

Department of Computer Science, Stony Brook University, USA

Abstract. Relationship-based access control (ReBAC) extends attribute-based access control (ABAC) to allow policies to be expressed in terms of chains of relationships between entities. 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. This paper presents algorithms for mining ReBAC policies from information about entitlements together with information about entities. It presents the first such algorithms designed to handle *incomplete* information about entitlements, typically obtained from operation logs, and *noise* (errors) in information about entitlements. We present two algorithms: a greedy search guided by heuristics, and an evolutionary algorithm. We demonstrate the effectiveness of the algorithms on several policies, including 3 large case studies.

1 Introduction

In *relationship-based access control* (ReBAC), access control policies are expressed in terms of chains of relationships between entities. This increases expressiveness and often allows more natural policies. High-level access control policy models such as ABAC and ReBAC are becoming increasingly important, as policies become more dynamic and more complex. This is reflected in the widespread transition from access control lists (ACLs) to role-based access control (RBAC), and more recently in the ongoing transition from ACLs and RBAC to ABAC. High-level policy models allow concise policies and promise long-term cost savings through reduced management effort.

Policy mining algorithms automatically produce a "first draft" of a high-level policy from existing lower-level data. They promise to drastically reduce the cost for an organization to migrate from a legacy access control technology to a high-level policy model. There is a significant amount of research on role mining, and role mining is supported by several commercial products, including IBM Tivoli Access Manager and Oracle Identity Analytics. Research on ABAC policy mining is much younger but growing as adoption of ABAC increases [14,13,7,4].

Mining of ReBAC policies, expressed as object-oriented ABAC policies with path expressions, has been explored in recent work by Bui, Stoller, and Li [3]. They present two algorithms, called the *greedy algorithm* and the *evolutionary algorithm*, to mine a ReBAC policy from a set of entitlements, a class model, and an object model. An

[⋆] This material is based on work supported in part by NSF Grants CNS-1421893, and CCF-1414078, ONR Grant N00014-15-1-2208, and DARPA Contract FA8650-15-C-7561.

entitlement is represented as a tuple $\langle subject, resource, action \rangle$, indicating that *subject* is authorized to perform *action* on *resource*.

The *meaning* of a policy π, denoted $[\![\pi]\!]$, is the set of granted entitlements. Both algorithms produce mined policies whose meaning is exactly the given set of entitlements. Consequently, they do not handle cases where that set is incomplete or noisy, which is often the case in practice. We propose more practical policy mining algorithms that do.

Incomplete information about entitlements is often readily available from operation logs, even when complete information is not, e.g., because the policy is not enforced by software, or because the policy is expressed using obscure *ad hoc* code. Many systems produce operation logs, e.g., for auditing or accounting. A set of entitlements can easily be extracted from a log. However, that set is typically *incomplete*, i.e., lacks some entitlements granted by the policy, because those entitlements were not exercised during the period covered by the log. We refer to these as *missing entitlements*. If the log contains entries for access requests that were denied, a set of *denials* can also be extracted from it. We represent denials as 3-tuples, just like entitlements.

Information about entitlements, even when nominally complete, is often noisy, i.e., contains errors in the form of *missing entitlements* (i.e., entitlements that should be present but aren't) and *excess entitlements* (i.e., entitlements that should not be present but are). Note that incomplete inputs typically have a much larger percentage of missing entitlements than noisy inputs.

We modify Bui et al.'s algorithms to handle incomplete and noisy inputs. Our algorithms identify suspected missing entitlements and add them to the meaning of the mined policy, so we also call them *added entitlements*. Our algorithms identify suspected excess entitlements and omit them from the meaning of the mined policy, so we also call them *omitted entitlements*.

To handle excess entitlements, our algorithms, like [14], construct a candidate policy, classify entitlements covered only by low-quality rules as suspected excess entitlements, and omit them from the meaning of the mined policy by discarding the low-quality rules.

We extend the greedy algorithm with two approaches for missing entitlements. The *validity-threshold (VT) approach* [14] modifies the algorithm to keep rules that are "almost valid", i.e., whose meaning contains a percentage of added entitlements that is below a specified threshold. The *extended quality (EQ) approach* [13] does not impose a strict cutoff on the percentage of added entitlements in the meaning of a rule. Instead, it extends the notion of rule quality with a term proportional to the percentage of added entitlements in the meaning of the rule, allowing a smooth trade-off between this and other aspects of quality. We refer to the versions of the greedy algorithm extended with these approaches as *VT greedy algorithm* and *EQ greedy algorithm*, respectively.

We extend the evolutionary algorithm to handle missing entitlements using a combination of the above approaches. We modify the fitness function so that, when the fraction of added entitlements in a rule's meaning is below a threshold, the added entitlements do not affect the rule's fitness, and when that fraction is above the threshold, the rule's fitness is redued proportionally to that fraction. We tried simpler approaches, but they produced worse results. We also modify the algorithm to use a validity threshold when deciding whether to add a candidate rule to the policy.

We evaluate our algorithms on four relatively small but non-trivial sample policies and three larger and more complex case studies. One sample policy is for electronic medical records (EMR), based on the EBAC policy in [2], translated to ReBAC; the other three are for healthcare, project management, and university records, based on ABAC policies in [14], generalized and made more realistic by translation to ReBAC. Two of the case studies are based on Software-as-a-Service (SaaS) applications offered by real companies [5,6]; one is based on a university's grant proposal workflow management system [10]. More details about these policies (other than the last one, which is new) appear in [3]. Our evaluation methodology is to start with a ReBAC policy π_0, compute the set $[\![\pi_0]\!]$ of granted entitlements, create from it a set of entitlements E_0 that is either incomplete (by pseudorandomly removing a significant percentage of the entitlements) or noisy (by pseudorandomly adding and removing small percentages of entitlements), run a policy mining algorithm on E_0 (along with the class model and object model from π_0), and compare the meaning $[\![\pi]\!]$ of the mined ReBAC policy π with $[\![\pi_0]\!]$. If the algorithm correctly compensates for the incompleteness or noise, they will be the same.

In our experiments with the greedy algorithm, the VT approach achieves better results than the EQ approach. We initially expected the EQ approach to be superior, because its quality metric is sensitive to the exact number of added entitlements. We now believe that the VT approach achieves better results because the goal is not to minimize the added entitlements, but rather to add just the right ones.

In our experiments comparing the VT greedy algorithm with the evolutionary algorithm, the VT greedy algorithm runs faster and achieves slightly to moderately better results. This is somewhat surprising, considering that, in Bui et al.'s experiments for ReBAC policy mining without incompleteness or noise [3], the evolutionary algorithm achieved somewhat better results than the greedy algorithm. It will be interesting to see if one of the algorithms can be improved to match or beat the other in both settings. One reason for favoring improvement of the evolutionary algorithm (e.g., by experimenting with new mutations) is its superior extensibility: it relies less on language-specific heuristics and hence is easier to extend to handle additional policy language features, e.g., additional data types (numbers, sequences, etc.) and associated relational operators. We plan to investigate extensibility in future work. We also plan to try to get real-world logs and associated policies to further evaluate our approach.

We also evaluated Rhapsody's approach to handling missing entitlements in the context of mining ABAC policies from logs [4]. There is no easy way to combine Rhapsody's approach with our algorithms, so we evaluate Rhapsody by running it and Xu and Stoller's EQ greedy algorithm for mining ABAC policies from logs [13] on the same data sets. We find that Rhapsody is much slower and would be usable for ReBAC mining only on small problem instances.

2 Policy Language

Since our algorithms are based on Bui et al.'s, we also adopt their ReBAC policy language, ORAL (Object-oriented Relationship-based Access-control Language)[3]. It formulates ReBAC as an object-oriented extension of ABAC. Relationships are expressed using attributes that refer to other objects, and path expressions are used in conditions and constraints to follow chains of relationships between objects. We describe the language briefly and refer the reader to [3] for details.

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. Each field has a *multiplicity* that specifies how many values may be stored in the field and is "one" (also denoted "1"), "optional" (also denoted "?"), or "many" (also denoted "*", meaning any number). Boolean fields always have multiplicity 1. Every class implicitly contains a field "id" with type String and multiplicity 1. 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. Let type(o) denote the type of object o. The value of a field with multiplicity "many" is a set. The value of a field with multiplicity "optional" may be a single value or the placeholder $\perp$ indicating absence of a value.

A *path* is a sequence of field names, written with "." as a separator. A *condition* is a set, interpreted as a conjunction, of atomic conditions. 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. For example, an object o satisfies $\langle \text{dept.id}, \text{in}, \{\text{CompSci}\}\rangle$ if the value obtained starting from o and following (dereferencing) the dept field and then the id field equals CompSci.

A *constraint* is a set, interpreted as a conjunction, of atomic constraints. Informally, an atomic constraint expresses a relationship between the requesting subject and the requested resource, by relating the values of paths starting from each of them. 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. 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. For example, a subject s and resource r satisfy $\langle \text{specialties}, \text{contains}, \text{topic}\rangle$ if the set s.specialties contains the value r.topic.

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. 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 $\text{acts}(\rho) = A$.

For readability, we may prefix paths with "subject" or "resource", to indicate the object from which the path starts. For example, our e-document case study involves a large bank whose policy contains the rule: A project member can read all sent documents regarding the project. This is expressed as $\langle$ Employee, subject.employer.id = LargeBank, Document, true, subject.workOn.relatedDoc $\ni$ resource,{read}$\rangle$, where Employee.workOn is the set of projects the employee is working on, and Project.relatedDoc is the set of sent documents related to the project.

The *type of a path p* (relative to a specified class), denoted type(p), is the type of the last field in the path. Given a class model, object model, object o, and path p, let nav(o, p) be the result of navigating (a.k.a. following or dereferencing) path p starting from object o. The result might be no value, represented by $\perp$, an atomic value, or (if a field in p has multiplicity many) a set of values. This is like the semantics of path navigation in UML's Object Constraint Language.

An object o *satisfies* an atomic condition $c = \langle p, op, val\rangle$, denoted $o \models c$, if $(op = \text{in} \wedge \text{nav}(o,p) \in val) \vee (op = \text{contains} \wedge \text{nav}(o,p) \ni val)$. Objects o_1 and o_2 *satisfy* an atomic constraint $c = \langle p_1, op, p_2\rangle$, denoted $\langle o_1, o_2\rangle \models c$, is defined in a similar way.An entitlement $\langle s, r, a\rangle$ *satisfies* a rule $\rho = \langle st, sc, rt, rc, c, A\rangle$, denoted $\langle s, r, a\rangle \models \rho$, 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 set of entitlements that satisfy it. The *meaning* of a ReBAC policy π, denoted $[\![\pi]\!]$, is the union of the meanings of its rules.

3 Problem Definition

A ReBAC policy that grants a given set E_0 of entitlements can be trivially constructed, by creating a separate rule that grants each entitlement in E_0, using conditions on the "id" field to specify the relevant subject and resource. Such a ReBAC policy is no better than ACLs.

We adopt two criteria to specify which ReBAC policies are most desirable. One criterion is to use "id" field only when necessary, i.e., only when every ReBAC policy consistent with π_0 contains rules that use it, because rules that use the "id" field are identity-based, not attribute-based or relationship-based. The other is to maximize a *policy quality metric*, which is a function Q_{pol} from ReBAC policies to a totally-ordered set, e.g., natural numbers. For generality, we parameterize the policy mining problem by the policy quality metric, with the convention that smaller values indicate higher quality.

The *extended ReBAC policy mining problem* is: given a set E_0 of entitlements, a set D_0 of denials, an object model OM, and a class model CM, find a set *Rules* of rules such that the ReBAC policy $\pi = \langle CM, OM, Act, Rules\rangle$ that uses the "id" field only when necessary, denies all requests in D_0, and has the best quality, according to Q_{pol}, among such policies. Here, *Act* is the set of actions that appear in E_0.

We call this the "extended" problem to distinguish it from the ReBAC policy mining problem in [3], which requires $[\![\pi]\!] = E_0$ and can be viewed as a special case corresponding to policy quality metrics that give overwhelming penalty to mismatches between $[\![\pi]\!]$ and E_0.

The policy quality metric that our algorithms aim to optimize is a sum of three terms. Our algorithms do not guarantee to optimize it; that is NP-hard even for ABAC mining [13]. The first term is *weighted structural complexity* (WSC), a generalization of policy size. It is the same as in [3]. Minimizing policy size is consistent with prior work on ABAC mining and role mining and with usability studies showing that concise policies are easier to manage [1]. The WSC of a policy, denoted WSC(π), is the sum of the WSCs of its rules. The WSC of a rule ρ, denoted WSC(ρ) is a weighted sum of the WSCs of its components, ignoring the two types, because they always have the same size. We ignore the weights hereafter, because we always set them to 1. 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 a condition is the sum of the WSCs of the atomic conditions in it. The WSC of a constraint is defined similarly [3]. The WSC of an action set is its cardinality.

The second and third terms measure differences between $[\![\pi]\!]$ and E_0. They are not needed in [3], which requires $[\![\pi]\!] = E_0$. They measure the numbers of addeed and omitted entitlements, respectively. We divide them by $|OM|$ (the number of objects), because incompleteness and noise are typically characterized by percentages, not absolute numbers, of affected entitlements, and the number of granted entitlements is typically proportional to the size of the object model.

In summary, policy quality is $Q_{\mathrm{pol}}(\pi) = w_{\mathrm{wsc}}\mathrm{WSC}(\pi) + w_{\mathrm{add}}|\,[\![\pi]\!] \setminus E_0|/|OM| + w_{\mathrm{omit}}|E_0 \setminus [\![\pi]\!]\,|/|OM|$, where the weights are user-specified.

4 Greedy Algorithm

Our VT and EQ greedy algorithms are based on the greedy algorithm for ReBAC policy mining (without incompleteness or noise) in [3]. It has three phases. The first phase iterates over the given entitlements, uses selected entitlements as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover more of the given entitlements, greedily selecting the highest-quality generalization according to a heuristic rule-quality metric. The second phase improves the policy by merging and simplifying candidate rules. The third phase selects the highest-quality candidate rules for inclusion in the mined policy.

4.1 Validity-threshold (VT) Approach

Top-level pseudocode appears in Figure 1. It returns a rule set Rules'. Entitlements covered by Rules' and not in E_0 are the suspected missing entitlements. Entitlements in E_0 not covered by Rules' are the suspected excess entitlements. It calls several functions, described below, after a summary of how missing and excess entitlements are handled. Note that function names hyperlink to descriptions of the functions.

Missing entitlements. The algorithm has a parameter α that bounds the acceptable fraction of added entitlements (i.e., entitlements not in E_0) for a rule. A rule ρ is α-valid iff the fraction of added entitlements is at most α (i.e., $|\,[\![\rho]\!] \setminus E_0| \div |\,[\![\rho]\!]\,| \leq \alpha$) and the rule does not cover any denials (i.e., $[\![\rho]\!] \cap D_0 = \emptyset$). The usual notion of *validity* [3] corresponds to $\alpha = 0$. Candidate rules are checked for α-validity at several places in the algorithm, as discussed below.

Excess entitlements. Excess entitlements typically result from individual errors in policy administration and hence do not fit any pattern that can be expressed concisely as rules. Consequently, excess assignments lead to the creation of low-quality candidate rules. As described in Section 1 and embodied in the last line in Figure 1, our algorithm drops rules whose quality is below a threshold τ, which is a parameter of the algorithm.

The function candidateConstraint(s, r) returns a set containing all the atomic constraints that hold between subject s and resource r and that satisfy specified limits on the lengths of the path expressions. It first computes all candidate constraints

```
// Phase 1: Create a set Rules of candidate rules that covers $E_0$.
$Rules = \emptyset$
// uncov contains entitlements in $E_0$ that are not covered by Rules
$uncov = E_0$.copy()
while $uncov$ is not empty
  // Use highest-quality uncovered entitlement as a "seed" for rule creation. Quality
  // of $\langle s, r, a\rangle$ is proportional to the number of occurrences in $E_0$ of $\langle r, a\rangle$ and $s$.
  $\langle s, r, a\rangle$ = highest-quality entitlement in $uncov$
  $cc$ = candidateConstraint($s, r$)
  // $s_{\mathrm{sub}}$ contains subjects with permission $\langle r, a\rangle$ and with same candidate constraints as $s$
  $s_{\mathrm{sub}} = \{s' \in OM \mid \mathrm{type}(s') = \mathrm{type}(s) \wedge \langle s', r, a\rangle \in E_0 \wedge \mathrm{candidateConstraint}(s', r) = cc\}$
  // Add candidate rule that covers at least permission $\langle r, a\rangle$ for subjects in $s_{\mathrm{sub}}$
  addCandidateRule(type($s$), $s_{\mathrm{sub}}$, type($r$), $\{r\}$, $cc$, $\{a\}$, $uncov$, $Rules$)
  // $s_{\mathrm{act}}$ is set of actions that $s$ can perform on $r$
  $s_{\mathrm{act}} = \{a' \in Act \mid \langle s, r, a'\rangle \in E_0\}$
  // Add candidate rule that covers at least permissions $\{\langle r, a'\rangle \mid a' \in s_{\mathrm{act}}\}$ for subject $s$
  addCandidateRule(type($s$), $\{s\}$, type($r$), $\{r\}$, $cc$, $s_{\mathrm{act}}$, $uncov$, $Rules$)
end while
// Phase 2: Merge and simplify rules.
Repeatedly call mergeRules($Rules$), simplifyRules($Rules$), and
mergeRulesInheritance($Rules$), until they have no further effect
// Remove redundant rules
while $Rules$ contains rules $\rho$ and $\rho'$ such that $[\![\rho]\!] \subseteq [\![\rho']\!]$
  $Rules$.remove($\rho$)
end while
// Phase 3: Select high quality rules into $Rules'$.
$Rules' = \emptyset$
Repeatedly move highest-quality rule from $Rules$ to $Rules'$ until $\sum_{\rho \in Rules'} [\![\rho]\!] \supseteq E_0$,
using $E_0 \setminus [\![Rules']\!]$ as second argument to $Q_{\mathrm{rul}}$, and discarding a rule if it does not
cover any tuples in $E_0$ currently uncovered by $Rules'$ or if its quality is below $\tau$
return $Rules'$
```

Fig. 1. Greedy algorithm for the extended ReBAC policy mining problem. Inputs: subject-permission relation E_0, class model CM, object model OM. Output: set of rules $Rules'$.

that contain type-correct paths that start from type(s) and type(r), respectively, and satisfy the length limits, and then it computes and returns the subset of these that are satisfied by $\langle s, a\rangle$. The length limits mainly bound the difference between the length of the path and the length of the shortest path between the same types.

The function addCandidateRule($st, s_{\mathrm{sub}}, rt, s_{\mathrm{res}}, cc, s_{\mathrm{act}}, uncov, Rules$) first computes conditions sc and rc that characterize (i.e., whose meaning equals) the set s_{sub} of subjects and the set s_{res} of resources, respectively. It tries to do this using paths that satisfy the length limits and without using the path "id" (this is a path of length 1); if this is insufficient, an atomic condition on the path "id" is added. addCandidateRule then constructs a rule $\rho = \langle st, sc, rt, rc, \emptyset, s_{\mathrm{act}}\rangle$, calls generalizeRule (described below) to generalize ρ to ρ', adds ρ' to candidate rule set $Rules$, and then removes the entitlements covered by ρ' from $uncov$.

The function generalizeRule(ρ, *cc*, *uncov*, *Rules*) attempts to generalize rule ρ by adding some atomic constraints in *cc* to ρ and eliminating the conjuncts (if any) of the subject condition and resource condition that use the same paths as those atomic 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, and its meaning is a superset of the meaning of ρ. In more detail, generalizeRule tries to generalize ρ using each constraint in *cc* separately, discards the generalizations that are not α-valid, sorts the α-valid generalizations in descending order of the number of covered entitlements in *uncov*, recursively tries to further generalize each of them using constraints from *cc* that produced α-valid generalizations later in the sort order, and then returns the highest-quality rule among them (rule quality is defined below); if no generalizations of ρ are α-valid, it simply returns ρ.

A *rule quality metric* is a function $Q_{\mathrm{rul}}(\rho, E)$ that maps a rule ρ to a totally-ordered set, with the order chosen such that larger values indicate higher quality. The second argument E 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 (because constraints tend to produce more general rules than conditions), and a tertiary preference for rules with shorter paths in constraints, we define $Q_{\mathrm{rul}}(\rho, E) = \langle |\,[\![\rho]\!] \cap E|/\mathrm{WSC}(\rho),\ |\mathrm{con}(\rho)|, 1/\mathrm{TCPL}(\rho)\rangle$ where $\mathrm{TCPL}(\rho)$ ("total constraint path length") is the sum of the lengths of the paths used in the constraints of ρ. In generalizeRule, the second argument to Q_{rul} is *uncov*, so $[\![\rho]\!] \cap E$ is the set of currently uncovered entitlements that are covered by ρ.

The function mergeRules(*Rules*) attempts to improve the quality of *Rules* by merging pairs of rules with the same subject type, resource type, and constraint by taking the least upper bound (LUB) of their subject conditions, the LUB of their resource conditions, and the union of their sets of actions. The *least upper bound* of conditions c_1 and c_2 is obtained by combining "in" conditions with the same path in c_1 and c_2, keeping "contains" conditions with the same path and constant in c_1 and c_2, and dropping other atomic conditions in c_1 and c_2. Thus, if c_1 contains $\langle p, \mathrm{in}, \mathit{val}_1\rangle$ and c_2 contains $\langle p, \mathrm{in}, \mathit{val}_2\rangle$, then their LUB contains $\langle p, \mathrm{in}, \mathit{val}_1 \cup \mathit{val}_2\rangle$; and, if c_1 contains $\langle p, \mathrm{contains}, \mathit{val}\rangle$ and c_2 contains $\langle p, \mathrm{contains}, \mathit{val}\rangle$, then their LUB contains $\langle p, \mathrm{contains}, \mathit{val}\rangle$. 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 ρ_{mrg} is α-valid, then it is added to *Rules*, and ρ_1 and ρ_2 are redundant and will be removed later.

The function simplifyRules(*Rules*) attempts to simplify each of the rules in *Rules* using several transformations, detailed in [3]. For example, it eliminates atomic conditions from the subject condition and resource condition, and eliminates atomic constraints from the constraint, if the resulting rule is α-valid.

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 resulting rule is α-valid, if any.

Complexity Analysis. The step with the highest asymptotic complexity is mergeRules, since the number of attempted merges is $O(|\mathit{Rules}|^2)$. In the worst case, each rule covers only one entitlement, and this is quadratic in log size. In practice, a rule covers many entitlements on average, and the complexity is much less. The complexity is similar as for Bui et al.'s algorithm [3], and in their experiments, measured growth in running time is less than quadratic with respect to number of entitlements.

Example. We illustrate the VT greedy algorithm on a small fragment of the workforce management case study containing only the rule "Help desk operators can modify work orders that apply to active contracts of a Primary Tenant for which he/she is assigned responsible", formalized as $\rho_0 = \langle$HelpdeskOperator, true, WorkOrder, resource.contract.active = true, subject.tenants $\ni$ res.contract.tenant,{modify}$\rangle$.

The object model contains: PrimaryTenant objects telco and pp; HelpdeskOperator objects ho1 and ho2 with ho1.tenants ={telco, pp} and ho2.tenants ={pp}; Contract objects telcoActive, telcoInactive, and ppActive whose tenant and active status are as indicated in the name; WorkOrder objects telcoWO1 on telcoActive contract, telcoWO2 on telcoInactive contract, and ppWO1 on ppActive contract.

For the policy π containing only ρ_0, $[\![\pi]\!] = \{\langle \mathrm{ho1}, \mathrm{telcoWO1}, \mathrm{modify}\rangle, \langle \mathrm{ho1}, \mathrm{ppWO1}, \mathrm{modify}\rangle, \langle \mathrm{ho2}, \mathrm{telcoWO1}, \mathrm{modify}\rangle\}$. Suppose the input E_0 is missing entitlement $\langle ho1, ppWO1, modify\rangle$ and contains excess entitlement $\langle ho2, ppWO1, modify\rangle$.

Our algorithm selects $\langle \mathrm{ho2}, \mathrm{telcoWO1}, \mathrm{modify}\rangle$ as the first seed. The first call to addCandidateRule creates a rule with conditions subject.tenants $\ni$\{Telco\}, resource.contract.tenant $\in$\{Telco\}, and resource.contract.active = true, and then calls generalizeRule on it. The generalization with candidate constraint subject.tenants $\ni$ resource.contract.tenant succeeds, removing the first two conditions and creating a rule ρ_1 identical to ρ_0, provided $\alpha \geq 1/3$ (to allow covering the missing entitlement). The second call to addCandidateRule generates a rule similar to rule ρ_0 except that it has additional condition subject = ho2; later, this rule is merged with ρ_1, and the merge leaves ρ_1 unchanged.

Our algorithm selects $\langle \mathrm{ho2}, \mathrm{ppWO1}, \mathrm{modify}\rangle$ as the next seed. The two calls to addCandidateRule generate two rules without constraints; merging and simplification produces a rule with conditions subject ={ho2} and resource.contract.tenant ={pp}. This rule's quality is low, since it covers only 1 entitlement (the excess one). It will be discarded, provided $\tau \geq 1/6$.

4.2 Extended-Quality (EQ) Approach

The main difference in this approach compared to the VT approach is that the algorithm uses a modified rule quality metric that takes added entitlements into account. The rule quality metric Q_{rul} in Section 4.1 is replaced with a rule quality metric $Q_{\mathrm{rul}}^{\mathrm{EQ}}$ whose first component includes a factor that imposes a penalty for added entitlements, measured as a fraction of the number of entitlements covered by the rule, and with a weight specified by a parameter $w_{\mathrm{rul}}^{\mathrm{EQ}}$.

$$Q_{\mathrm{rul}}^{\mathrm{EQ}}(\rho, E) = \langle \frac{|\,[\![\rho]\!] \cap E|}{\mathrm{WSC}(\rho)} \times (1 - \frac{w_{\mathrm{rul}}^{\mathrm{EQ}} \times |\,[\![\rho]\!] \setminus E_0|}{|\,[\![\rho]\!]\,|}), |\mathrm{con}(\rho)|, 1/\mathrm{TCPL}(\rho)\rangle$$

Also, the four functions that involve α-validity checks are modified as follows. In generalizeRule, the α-validity check is replaced with a check that the rule does not cover any tuples in D_0. In mergeRules, instead of checking whether ρ_{mrg} is α-valid, the algorithm compares the policy quality (as defined in Section 3, with $w_{\text{omit}} = 0$, since omitted entitlements are not determined yet) of *Rules* and $\mathit{Rules} \cup \{\rho_{\text{mrg}}\} \setminus \{\rho_1, \rho_2\}$, where ρ_1 and ρ_2 are the rules being merged. If the latter has higher quality, and ρ_{mrg} does not accept any tuples in D_0, then ρ_1 and ρ_2 are replaced with ρ_{mrg}. In simplifyRules(*Rules*) and mergeRulesInheritance(*Rules*), instead of checking α-validity of the simplified or merged rule, the algorithm checks that the rule does not cover any denials in D_0 and that it does not cover any *new* added entitlements, i.e., entitlements not in E_0 and not currently covered by *Rules*; allowing new added entitlements in those places led to overly permissive policies.

5 Evolutionary Algorithm

Our evolutionary algorithm is based on the evolutionary algorithm for ReBAC mining (without incompleteness or noise) in [3], which is inspired by Medvet et al.'s work [7]. It is in the context-free grammar genetic programming (CFGGP) paradigm, in which individuals, which in our context are ReBAC rules, are represented as derivation trees of a context-free grammar (CFG). The main part of the algorithm is preceded by *grammar generation*, which specializes the generic grammar of ORAL to a specific input, so that rules in the language of the grammar contain only classes, fields, constants, and actions that appear in the input, and all path expressions are type-correct and satisfy the same length limits as in the greedy algorithm.

The algorithm's first phase iterates over the given entitlements, and uses each of the selected entitlements as the seed for an evolutionary search that adds one new rule to the candidate policy. Each evolutionary search starts with an initial population containing candidate rules created from a seed tuple in a similar way as in the greedy algorithm along with numerous random variants of those rules together with some completely random candidate rules, evolves the population by repeatedly applying genetic operators (mutations and crossover), and then selects the highest quality rule in the population as the result of that evolutionary search. The second phase improves the candidate rules by further mutating them.

Pseudocode appears in Figure 2. Function initialPopulation($\langle s, r, a\rangle$, *Rules*, *uncov*) creates an initial population for the evolutionary search for a high-quality rule that covers the seed $\langle s, r, a\rangle$ and other tuples. It is implicitly parameterized by the desired population size *popSize*. Half of the initial population is generated as follows: perform the same two calls to addCandidateRule as in Figure 1, add those rules to the initial population, and then add random variants of those rules obtained by removing some atomic conditions and atomic constraints. The other half consists of rules with subject type type(s) or one of its ancestors (selected randomly), resource type type(r) or one of its ancestors, randomly generated conditions and constraint, and action set $\{a\}$.

Rule quality is measured using a fitness function f, modified from the one in [3] to take missing assignments into account using a threshold approach: false acceptances are ignored unless they exceed a threshold specified by algorithm parameter α. The fitness function is $f(\rho, \alpha) = \langle \text{FAR}(\rho, \alpha), \text{FRR}(\rho), \text{ID}(\rho), \text{WSC}(\rho)\rangle$. The *false acceptance rate* FAR(ρ, α) is 0 if $\frac{|[\![\rho]\!] \setminus E_0|}{|[\![\rho]\!]|} < \alpha$ and is $\frac{|[\![\rho]\!] \setminus E_0|}{|[\![\rho]\!]|}$ otherwise. The *false rejection*

```
// Phase 1: Construct candidate policy, using evolutionary search to create each rule.
$Rules = \emptyset$; $uncov = E_0$.copy()
while $uncov$ is not empty
  $seed$ = highest-quality entitlement in $uncov$ (same quality metric as in greedy algorithm)
  $pop$ = initialPopulation($seed$, $Rules$, $uncov$)
  for $gen = 1$ to $nGenerationsSearch$
    $op$ = a genetic operator randomly selected from $searchOps$
    $S$ = set of $nTournament$ rules randomly selected from $pop$
    if $op$ is a mutation
      $pop$.add(the rule generated by applying $op$ to the highest-quality rule in $S$)
    else    // op is a cross-over
      $pop$.add(the two rules generated by applying $op$ to the two highest-quality rules in $S$)
    end if
    remove the lowest-quality rules in $pop$ until $|pop| = popSize$
  end for
  $\rho$ = the highest-quality rule in $pop$
  if $\alpha$-valid($\rho$);
    $Rules$.add($\rho$); $uncov$.removeAll($[\![\rho]\!]$)
  end if
end while
// Phase 2: Improve the candidate rules by further mutating them.
for each $\rho$ in $Rules$
  for $gen = 1$ to $nGenerationsImprove$
    if $gen = nGenerationsImprove/2 \wedge$ (all attempted improvements to $\rho$ failed)
      break  // This rule is unlikely to improve. Don't bother trying more.
    end if
    $op$ = a genetic operator randomly selected from $improveOps$
    $\rho'$ = the rule generated by applying $op$ to $\rho$
    if $\alpha$-valid($\rho'$) $\wedge$ ID($\rho'$) $\leq$ ID($\rho$)
      $redundant = \{\rho_0 \in Rules \mid [\![\rho_0]\!] \subseteq [\![\rho']\!]\}$
      if ($Rules \cup \{\rho'\} \setminus redundant$) covers $E_0$ and has lower WSC than $Rules$
        $Rules$.removeAll($redundant$); $Rules$.add($\rho'$)
      end if
    end if
  end for
end for
Repeatedly call mergeRules($Rules$) and simplifyRules($Rules$) until they have no effect
return the rules in $Rules$ with quality at least $\tau$
```

Fig. 2. Evolutionary algorithm for extended ReBAC policy mining problem. Inputs: subject-permission relation E_0, class model CM, object model OM. Output: set of rules $Rules$.

rate is $\mathrm{FRR}(\rho) = |uncov \setminus [\![\rho]\!]|$, and $\mathrm{ID}(\rho)$ is the number of atomic conditions in ρ with path "id".

The two validity checks used in the algorithm in [3] are replaced with α-validity checks, as shown in Figure 2. Also, to deal with excess entitlements, the algorithm returns only rules with quality at least τ.

The set *searchOps* of genetic operators used in the search phase contains the two traditional CFGGP genetic operators: a mutation operator that randomly selects a non-terminal in the derivation tree being evolved, and replaces the existing subtree rooted at that non-terminal with a new subtree randomly generated starting from that non-terminal, and a cross-over operator that randomly selects a non-terminal that appears in both of the derivation trees being evolved (called "parents"), and swaps the subtrees rooted at that non-terminal. It also contains a *double mutation* operator that mutates two out of the three predicates (the subject condition, resource condition, and constraint) in a rule (this enables the operator to have an effect similar to generalizeRule), and a *simplify mutation* that removes one randomly selected atomic condition or atomic constraint (this mutation is included to increase the overall probability of these mutations). The set *improveOps* of genetic operators used in the improvement phase is similar, except it also contains a *type mutation* operator that can replace the subject type or resource type with one of its ancestors.

The version of simplifyRules used in this algorithm is the same as in the greedy algorithm except extended with an additional simplification: replace the subject type or resource type with one of its children, if the policy still covers E_0.

6 Evaluation

This section presents experimental results evaluating our algorithms on the four sample policies and three large case studies mentioned Section 1, following the evaluation methodology sketched in Section 1. Our code and data are available at `http://www.cs.stonybrook.edu/~stoller/software/`. Parameters of the algorithms (e.g., *popSize*) have the same values as in [3]. Since the results of experiments with the EMR and project management sample policies are similar to the results for the other two sample policies, we summarize their results, omitting details due to space constraints. Each policy has handwritten class model and rules, and a synthetic object model generated by a policy-specific pseudorandom algorithm designed to produce realistic object models, by creating objects and selecting their attribute values using appropriate probability distributions.

Similarity Metrics. We evaluate the quality of the generated policy using three similarity metrics. They are normalized to range from 0 (completely different) to 1 (identical). They are based on *Jaccard similarity* of sets, defined by $J(S_1, S_2) = |S_1 \cap S_2| \,/\, |S_1 \cup S_2|$. The *semantic similarity* of policies π_1 and π_2 is $J([\![\pi_1]\!], [\![\pi_2]\!])$. We use this metric to compare meaning of the original policy π_0 and the mined policy π. If the algorithm accurately identifies and compensates for all incompleteness and noise, the semantic similarity will equal 1. *Missing entitlements similarity* is the Jaccard similarity of the set of actual missing entitlements (removed when creating E_0 from $[\![\pi_0]\!]$) and the set of suspected missing entitlements. *Excess assignments similarity* is the Jaccard similarity of the actual excess entitlements (added when creating E_0 from $[\![\pi_0]\!]$) and the suspected excess entitlements.

Running Time. An organization needs to run policy mining occasionally, not frequently, so our evaluation focuses on quality of results. Both algorithms have reasonable running times, although the VT greedy algorithm is significantly faster than the evolutionary algorithm. With our implementation in Java on an Intel i7-3770 CPU, each run of the VT greedy algorithm and evolutionary algorithm take at most 8.5 and 70 minutes, respectively. The policies involve up to several hundred objects, a few thousand entitlements, and a few dozen rules (more details on policy size are in [3]). The evolutionary algorithm can be sped up significantly, at the cost of a small decrease in quality, by varying parameters. For example, for e-document (our largest case study), reducing the number of generations per evolutionary search from 2000 (the value used in our main experiments) to 1000 reduces the running time by 27%, with a decrease of only 0.02 (from 0.87 to 0.85) in policy semantic similarity.

6.1 Experiments with Noise

We introduce synthetic noise at a specified level into the meaning of a ReBAC policy π_0 in a similar way as [14], apply our policy mining algorithms to the resulting set of entitlements E_0 along with the class model and object model, and then compute the above metrics comparing the original policy π_0 and mined policy π. Noise level is expressed as a fraction of $|[\![\pi_0]\!]|$; thus, noise level ν means that $\nu|[\![\pi_0]\!]|$ entitlements are added to or removed from $[\![\pi_0]\!]$. To introduce a specified level ν of noise, we introduce $\nu|[\![\pi_0]\!]|/6$ missing entitlements and $5\nu|[\![\pi_0]\!]|/6$ excess entitlements. This ratio is based on the data in [8, Table 1]. Missing entitlements are selected from a discrete normal distribution on $[\![\pi_0]\!]$, to reflect that policy errors are usually non-uniformly distributed. Excess entitlements are selected from a discrete normal distribution on the complement of $[\![\pi_0]\!]$. We tune all of the algorithm parameters manually, so the experimental results reflect the capabilities of the algorithms with an experienced user.

Figure 3 shows results for the VT greedy algorithm and evolutionary algorithm. Each datapoint is the average over 10 runs (except 5 runs for grant proposal and e-doc) on inputs with different pseudorandom object model and noise. The 95% confidence intervals using Student's t-distribution are reasonably small, less than 0.13 in all cases except missing assignment similarity for grant proposal policy when running with evolutionary algorithm, for which it is 0.18. For experiments on healthcare and university sample policies (not shown in the figure), both algorithms achieve perfect values (i.e., 1.0) on all three similarity metrics at all three noise levels.

To compare the algorithms, we average the similarity metrics over all three noise levels and all seven policies. For both algorithms, the average policy semantic similarity is 0.99, and the average excess entitlement similarity is 0.98; the latter is not surprising, since the algorithms use the same approach to identify excess entitlements. The average missing entitlement similarity is 0.96 for the greedy algorithm, and 0.94 for the evolutionary algorithm. We conclude that VT greedy algorithm is slightly better than evolutionary algorithm at detecting missing entitlements. Noise detection results for the EQ greedy algorithm are significantly worse: the average excess entitlement similarity is 0.95, and the average missing entitlement similarity is 0.73.

6.2 Experiments with Incompleteness (Mining from Logs)

Given a set of entitlements E_0, which is a subset of the meaning of a ReBAC policy π_0, the *completeness* of E_0 (relative to π_0) is $|E_0|/|[\![\pi_0]\!]|$, i.e., the fraction of entitlements

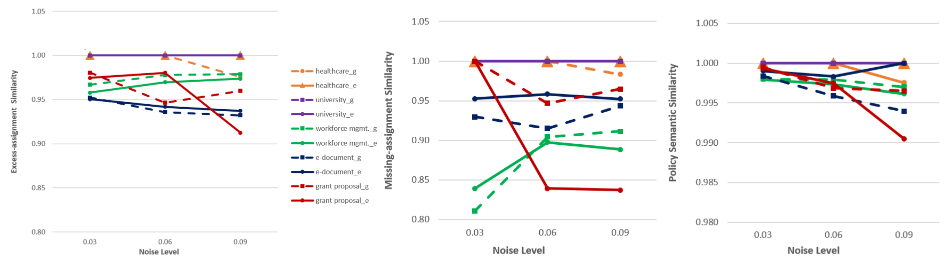

Fig. 3. Left: Excess entitlement similarity. Center: Missing entitlement similarity. Right: Policy semantic similarity. The legend is the same for all three graphs. Suffixes "_g" and "_e" indicate VT greedy algorithm and evolutionary algorithm, respectively. Results for the two algorithms are plotted with dashed and solid lines, respectively.

in $[\![\pi_0]\!]$ that are in E_0. Given a ReBAC policy π_0 and a desired completeness level c, we pseudorandomly select $(1-c)|\,[\![\pi_0]\!]\,|$ entitlements in $[\![\pi_0]\!]$ and remove them to create E_0. We select them from a discrete normal distribution on $[\![\pi_0]\!]$, to reflect that some entitlements are used more often and hence more likely to appear in an access log. We also generate a set of denials D_0 by pseudorandomly selecting tuples from a discrete normal distribution on the complement of $[\![\pi_0]\!]$. We set the number of denials to 4% of $|\,[\![\pi_0]\!]\,|$, based on Cotrini et al.'s comment that the percentage of denied operations in logs used in their experiments is usually less than 5% [4]. For simplicity, we do not add excess entitlements as noise in these experiments, because sets of entitlements obtained from logs are expected to contain a relatively small percentage of excess entitlements, which would not appreciably affect our results.

Figure 4 shows results for the VT greedy algorithm and evolutionary algorithm. Each datapoint is the average over 10 runs (except 5 runs for grant proposal and e-doc) on inputs with different pseudorandom object model and incompleteness. The 95% confidence intervals using Student's t-distribution are reasonably small, less than 0.12 in all cases. As expected, the results are better for higher completeness. For inputs with completeness 0.7 and higher, policy semantic similarity is above 0.93, and missing entitlements similarity is above 0.8, for all policies. For the healthcare and university sample policies (not shown in the figure), for all four completeness levels, both algorithms achieve perfect values on both similarity metrics.

To compare the algorithms, we average the similarity metrics over all four completeness levels and all seven policies. The average policy semantic similarity is 0.98 for VT greedy algorithm, and 0.97 for evolutionary algorithm. The average missing assignments similarity is 0.94 for VT greedy algorithm, and 0.91 for evolutionary algorithm. Thus, the VT greedy algorithm is modestly better than the evolutionary algorithm on incomplete inputs. Since the EQ greedy algorithm was less effective in the noise experiments, we run incompleteness experiments for it on a subset of policies, namely, EMR, grant proposal and workforce management. The results are consistent with our expectation: the average policy similarity is 0.89, compared to 0.97 for VT greedy algorithm on the same datasets, and the average missing assignments similarity is 0.67, compared to 0.89 for VT greedy algorithm on the same datasets.

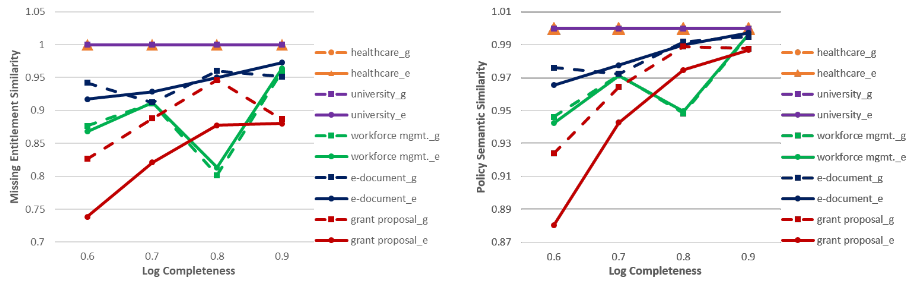

Fig. 4. Left: missing entitlements similarity. Right: policy semantic similarity. The suffixes and line styles have the same meaning as in Figure 3.

6.3 Comparison with Rhapsody

We evaluated Rhapsody's approach to handling incompleteness by running Rhapsody [4] and Xu et al.'s algorithm [13] on some of the ABAC policies used in [13]. For the university policy with manually written attribute data [13], which involves 10 ABAC rules and 16 attributes, and a log with completeness 0.8, Rhapsody's running time (based on its progress indicator) would exceed 24 hours. In contrast, Xu et al.'s algorithm produces a policy with a perfect policy semantic similarity of 1 in less than 1 second. We created a very small version of the policy, with only 7 ABAC rules and 9 attributes. After parameter tuning based on guidance in [4], the best result is a policy with policy semantic similarity 0.65, and Rhapsody took 3.7 hours to produce it.

The implementation of Rhapsody we were given considers neither conditions involving set relations nor constraints of any kind. Extending the implementation to consider these (as our algorithms do) would significantly increase the number of atoms (predicates that can appear in rules) and hence the running time. Further extending it to support ReBAC instead of ABAC would further greatly increase the number of atoms and hence the running time, making it usable only on small problem instances.

7 Related Work

The only prior work on mining of ReBAC policies (or object-oriented ABAC policies with path expressions) is [3], which is discussed in Section 1. The contributions of this paper include adapting their algorithms to handle incompleteness and noise using two approaches, and extensive experimental evaluation of the accuracy and performance of the two approaches and the two algorithms in this context. Interestingly, we find that the VT greedy algorithm achieves slightly to moderately better results; in contrast, the evolutionary algorithm achieves somewhat better results in the experiments in [3].

The earliest work on mining of access control policies from logs is Molloy et al.'s algorithm to mine "meaningful" roles from logs and attribute data, i.e., roles whose membership is statistically correlated with user attributes [9]. Their algorithm is based on a reduction to the author-topic model problem. Xu and Stoller adapted that approach to ABAC policy mining and found that it is less accurate and less scalable than the validity-threshold approach [13]. Other work on mining roles from incomplete or noisy data, e.g., [11,12], uses thresholds but does not consider attribute data.

References

1. Beckerle, M., Martucci, L.A.: Formal definitions for usable access control rule sets—From goals to metrics. In: Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS). pp. 2:1–2:11. ACM (2013)
2. Bogaerts, J., Decat, M., Lagaisse, B., Joosen, W.: Entity-based access control: supporting more expressive access control policies. In: Proc. 31st Annual Computer Security Applications Conference (ACSAC). pp. 291–300. ACM (2015)
3. Bui, T., Stoller, S.D., Li, J.: Greedy and evolutionary algorithms for mining relationship-based access control policies. Computers & Security (in press), also available at `http://arxiv.org/abs/1708.04749`. An earlier version appeared as a short paper in ACM SACMAT 2017.
4. Cotrini, C., Weghorn, T., Basin, D.: Mining ABAC rules from sparse logs. In: Proc. 3rd IEEE European Symposium on Security and Privacy (EuroS&P). pp. 2141–2148 (2018)
5. Decat, M., Bogaerts, J., Lagaisse, B., Joosen, W.: The e-document case study: functional analysis and access control requirements. CW Reports CW654, Department of Computer Science, KU Leuven (February 2014)
6. Decat, M., Bogaerts, J., Lagaisse, B., Joosen, W.: The workforce management case study: functional analysis and access control requirements. CW Reports CW655, Department of Computer Science, KU Leuven (February 2014)
7. Medvet, E., Bartoli, A., Carminati, B., Ferrari, E.: 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, pp. 351–365. Springer (2015)
8. Molloy, I., Li, N., Qi, Y.A., Lobo, J., Dickens, L.: Mining roles with noisy data. In: Proc. 15th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 45–54. ACM (2010)
9. Molloy, I., Park, Y., Chari, S.: Generative models for access control policies: applications to role mining over logs with attribution. In: Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT). ACM (2012)
10. Munakami, M.: Developing an ABAC-Based Grant Proposal Workflow Management System. Master's thesis, Boise State University (December 2016)
11. Vaidya, J., Atluri, V., Guo, Q., Lu, H.: Role mining in the presence of noise. In: Proc. 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec). Lecture Notes in Computer Science, vol. 6166, pp. 97–112. Springer (2010)
12. Vavilis, S., Egner, A.I., Petkovic, M., Zannone, N.: Role mining with missing values. In: Proc. 11th Int'l. Conference on Availability, Reliability and Security, (ARES) (2016)
13. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies from logs. In: Proc. 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec). Springer (2014), extended version available at `http://arxiv.org/abs/1403.5715`
14. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies. IEEE Transactions on Dependable and Secure Computing **12**(5), 533–545 (Sep–Oct 2015)