COMPUTERS & SECURITY 80 (2019) 317–333

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Greedy and evolutionary algorithms for mining relationship-based access control policies☆

Thang Bui, Scott D. Stoller, Jiajie Li*

Department of Computer Science, Stony Brook University, USA

ARTICLE INFO

Article history:
Received 30 April 2018
Revised 21 August 2018
Accepted 28 September 2018
Available online 6 October 2018

Keywords:
Access control policy mining
Relationship-based access control
Attribute-based access control
Evolutionary algorithms
Access control policy development

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 two algorithms for mining ReBAC policies from access control lists (ACLs) and attribute data represented as an object model: a greedy algorithm guided by heuristics, and a grammar-based evolutionary algorithm. An evaluation of the algorithms on four sample policies and two large case studies demonstrates their effectiveness.

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) (Gates, 2007). 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 2 compares ORAL with previous ReBAC models.

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 attribute-based access control (ABAC). In industry, more and more products support ABAC, using a standardized ABAC language such as XACML[1] (eXtensible Access Control Markup Language) or a vendor-specific ABAC language. In government, the Federal Chief Information Officer Council called out ABAC

☆ 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.

* Corresponding author.
E-mail address: stoller@cs.stonybrook.edu (S.D. Stoller).

[1] http://www.oasis-open.org/committees/xacml/.

https://doi.org/10.1016/j.cose.2018.09.011
0167-4048/

as a recommended access control model (Federal Chief Information Officer Council, 2011; Hu et al., 2013). ABAC allows "an unprecedented amount of flexibility and security while promoting information sharing between diverse and often disparate organizations" (Hu et al., 2013). ABAC and ReBAC overcome some of the problems associated with RBAC, notably role explosion (Hu et al., 2013), which makes RBAC policies large and hard to manage. High-level policy models allow concise policies and promise long-term cost savings through reduced management effort.

The cost of manually developing an initial high-level policy is a barrier to adoption of high-level policy models (Hu et al., 2013). *Policy mining* algorithms promise to drastically reduce this cost, by partially automating the process. Role mining, i.e., mining of RBAC policies, is an active research area (e.g., Alessandro et al., 2009; Alessandro et al., 2012; Barsha et al., 2016a; Barsha et al., 2016b; Haibing et al., 2008; Ian et al., 2010; Jaideep et al., 2006; Jaideep et al., 2010; Jaideep et al., 2007; Jürgen and Ulrike, 2005; Ludwig and Günther, 2008; Mario et al., 2013; Martin et al., 2003; Nino Vincenzo et al., 2012; Qi et al., 2008; Stoller and Thang, 2017; Xu and Stoller, 2012; Xu and Stoller, 2013) and a relatively small (about $70 million as of 2012) but rapidly growing commercial market segment (Safaá et al., 2012). Role mining is supported by several commercial products, including CA Technologies Identity Governance, Courion RoleCourier, IBM Tivoli Access Manager, Oracle Identity Analytics, NEXIS contROLE, and Novell Access Governance Suite. Research on ABAC policy mining is in the early stages, with initial work by Xu and Stoller (2014a,b, 2015) and Medvet et al. (2015). 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 algorithms 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 Xu and Stoller's proof that ABAC policy mining is NP-hard (Xu and Stoller, 2015). Since we desire an efficient and practical algorithm, our algorithms are not guaranteed to generate an optimal policy.

Our *greedy algorithm*, based on Xu and Stoller's algorithm for mining ABAC policies from ACLs (Xu and Stoller, 2015), 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 size of the rule. 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.

Our *evolutionary algorithm*, inspired by Medvet et al.'s evolutionary algorithm for mining ABAC policies, uses grammar-based genetic programming (McKay et al., 2010; Whigham., 1995). It has two phases. In the first phase, it iterates over tuples in the subject-permission relation, and uses each of the selected tuples 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 our 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 attempts to improve the candidate rules by further mutating them.

We evaluate our algorithms on four relatively small but non-trivial sample policies and two larger and more complex case studies, based on Software-as-a-Service (SaaS) applications offered by real companies (Decat et al., 2014a,b). To the best of our knowledge, the latter are the largest rule-based policies (as measured by the number and complexity of the rules) used in the evaluation of any policy mining algorithm targeting a rule-based policy language.

Our evaluation methodology is to start with a ReBAC policy, generate ACLs representing the subject-permission relation, run a policy mining algorithm on the generated ACLs (along with the class model and object model), and compare the ReBAC policy mined from ACLs with the original ReBAC policy. For the four sample policies, both of our policy mining algorithms achieve optimal or nearly optimal results. For the case studies, our greedy algorithm and evolutionary algorithm achieve 84% and 91% (respectively) average syntactic similarity between the mined policy and a simplified but equivalent version of the original policy. Experiments on object models of varying size for the two case studies show that both algorithms have good performance and scale reasonably well: as a function of the number of subject-permission tuples, the running time of the greedy algorithm is less than quadratic, and the running time of the evolutionary algorithm is close to linear.

We conclude that both algorithms produce high-quality mined policies that, if used as a starting point for development for a ReBAC policy, would save the policy developers a significant amount of effort.

This paper is a revised and greatly extended version of a SACMAT 2017 short paper (Thang et al., 2017), which briefly described an earlier version of our greedy algorithm and presented experimental results for it. The most significant addition in this paper is our evolutionary algorithm, and the experimental results comparing the effectiveness and performance of our two algorithms. Other notable additions are more detailed description of our greedy algorithm, an example that illustrates the working of our greedy algorithm, and more detailed descriptions of the sample policies and case studies.

2. Related work

We discuss related work on policy models and related work on policy mining.

2.1. Policy models

Entity-Based Access Control (EBAC) (Jasper et al., 2015) 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 et al. (2009), Fong (2011), Cheng et al. (2012), Hu et al. (2013), Crampton and Sellwood (2014), 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 relating fields (a.k.a. attributes) of different entities, such as the constraint subject.affiliation $\in$ resource.patient.registrations in the ORAL rule in Eq. (1). In this regard, ORAL is significantly more expressive.

On the other hand, ORAL lacks some features found in some of these ReBAC models, such as transitive closure (e.g., supervisor* refers to the subject's supervisor, the subject's supervisor's supervisor, and so on), negation (e.g., dept $\neq$ CS), and graph patterns (which can specify more than a single path). Many realistic applications do not require these language features; on the other hand, 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. That challenge is future work.

The languages in Fong (2011), Cheng et al. (2012), Glenn et al. (2012) and Crampton and Sellwood (2014) 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).

Our access control model can be characterized as *object-oriented ABAC*. We prefer to characterize it as ReBAC to emphasize the difference from typical ABAC languages such as XACML. In XACML, attributes values are primitive values, such as numbers or strings, or collections thereof, not object references. Primitive values can be (and often are) object identifiers, but they cannot be dereferenced, so policies that require path expressions cannot be expressed. For example, XACML can express the condition user.dept=CS but not user.dept.college=ENG. This limitation is typically circumvented by duplicating information, e.g., introducing an attribute user.college. This workaround is inefficient and increases the administrative burden, because user.college must be updated whenever user.dept is updated; in our framework, user.dept.college automatically has the correct value after user.dept is updated. Next Generation Access Control (NGAC) is an ABAC standard being developed at NIST (David et al., 2016). Its data model is richer than XACML's, allowing nested collections of entities, but it does not adopt a general object-oriented view in which subjects, resources, and other types of objects are modeled in a uniform way, and it does not support general path expressions.

2.2. Policy mining

There is no prior work on mining of ReBAC policies (or object-oriented ABAC policies with path expressions). 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 (Xu and Stoller, 2015), roles (Xu and Stoller, 2014b), or access logs (Xu and Stoller, 2014a). Our greedy algorithm is based on their algorithm for mining ABAC policies from ACLs (Xu and Stoller, 2015). 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 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 Xu and Stoller (2015); in constraints, we allow "in" in addition to "equal", "contains", and "supseteq" allowed in Xu and Stoller (2015). Other algorithm differences include deferring removal of redundant rules (by modifying mergeRules not to remove redundant rules, and removing redundant rules before final rule selection) and adding a third component to the rule quality metric. We also introduce several techniques to limit and prioritize the paths being considered, since naively considering all type-correct paths would make the algorithm prohibitively expensive, even for small policies. For example, when generating constraints, we base them only on the shortest and nearly-shortest paths between classes in the class model.

Medvet et al.'s evolutionary algorithm for ABAC policy mining (Medvet et al., 2015) inspired our evolutionary algorithm for ReBAC policy mining. Our algorithm, like theirs, has an evolutionary search phase using the separate-and-conquer strategy, followed by an improvement phase. The separate-and-conquer strategy (Alberto et al., 2015), which in the context of policy mining means learning one rule at a time, instead of an entire policy at once, is essential to obtain good results. We also adopt their fitness function, which, in turn, is based on Xu and Stoller's rule quality metric (Xu and Stoller, 2015). A key difference is that Medvet et al.'s algorithm uses an ad-hoc application-specific genotype (i.e., representation of individuals) together with genetic operators specifically designed to operate on that genotype. In contrast, we adopt the general and well-studied framework of grammar-based genetic programming (McKay et al., 2010; Whigham., 1995): we represent individuals as derivation trees, and we use genetic operators that operate on derivation trees. We use the classical mutation

and crossover operators on derivation trees but also introduce a few more genetic operators specialized to the general structure of our grammars (e.g., non-terminals for conditions and actions are treated differently), which enable the evolutionary search to produce high-quality rules much more quickly. The operators are not specific to details of the predicate language, so our algorithm can easily be applied to extensions of the policy language with additional datatypes and relational operators. Another difference from Medvet et al.'s algorithm is that our algorithm uses a more complicated construction for the initial population of each evolutionary search. These features enable our algorithm to achieve good results with reasonable computation time, despite the significantly larger search space for ReBAC policies compared with ABAC policies. Also, Medvet et al.'s algorithm was evaluated only on policies comparable in size to our four sample policies, not on large policies comparable to our two case studies.

Cotrini et al.'s algorithm for mining ABAC rules from sparse logs (Carlos et al., 2015) is based on APRIORI-SD, a machine-learning algorithm for subgroup discovery. "Sparse" means that only a small fraction of the possible entitlements appear in the log. Therefore, the algorithm must extrapolate significantly to determine which entitlements not in the log should be granted, and which should be denied. They formulate a novel heuristic to identify suspected over-permissiveness of ABAC rules. Their algorithm searches for succinct rules that have high confidence and are not overly permissive according to their heuristic.

3. Policy language

This section presents our policy language, ORAL. It contains common ABAC constructs, similar to those in Xu and Stoller (2015), 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 and by omitting other base types (e.g., numbers); they could easily be added. 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 field. 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. Let type(o) denote the type of an object o.

A *condition* is a set, interpreted as a conjunction, of atomic conditions. We often refer to the atomic conditions as conjuncts. 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 (specifically, a Boolean value or string) or a set of constants. If *val* is a single constant, not a set, we say that it is *atomic*. Note that *val* cannot equal or contain the placeholder $\perp$. A *path* is a sequence of field names, written with "." as a separator. For example, if dept and id are field names, then dept.id is a path. For readability, we usually write conditions with a logic-based syntax, 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. 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 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. 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. For readability, we usually write constraints with a logic-based syntax, 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.

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.

Examples. We give three example rules here. As additional examples, all of the sample policies and case studies described in Section 7 are available on the web, in the ReBAC Miner release at http://www.cs.stonybrook.edu/~stoller/software/. In examples, we prefix the path in the subject condition and resource condition with "subject" and "resource", respectively, for readability. Our electronic medical records sample policy

contains the rule: A physician can create a medical record associated with a consultation if the physician is not a trainee, the consultation is with the physician, and the patient of the consultation is registered at the hospital with which the physician is affiliated. This is expressed as

$$\rho = \langle \text{Physician, subject.isTrainee=false, Consultation, true, subject = resource.physician} \wedge \text{subject.affiliation} \in \text{resource.patient.registrations, \{createMedicalRecord\}}\rangle. \quad (1)$$

Our healthcare sample policy contains the rule: 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. This is 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. 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 ⟨Employee, subject.employer.id = LargeBank, Document, true, subject.workOn.relatedDoc ∋ resource, {read}⟩, where Employee.workOn is the set of projects the employee is working on, and Project.relatedDoc is the set of documents related to the project.

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∋topic.id, can be written more simply as specialties∋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 value is a set of constants. This excludes sets of sets from the model. (5) In conditions with operator "contains", the path has multiplicity many, and the value is atomic. (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.

Any class can be used as a subject type, resource type, or both. For example, one rule could allow doctors to read medical records, and another rule could allow department heads to assign doctors to workgroups.

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

A *permission* is a pair $\langle r, a\rangle$, where r is an object, and a is an action; it represents authorization to perform action a on resource r. A *subject-permission* tuple is a tuple $\langle s, r, a\rangle$, where s is an object, and $\langle r, a\rangle$ is a permission; it means that subject s has permission $\langle r, a\rangle$. A *subject-permission relation* is a set of subject-permission tuples.

Given a 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. The result of navigating might be no value, represented by the placeholder $\perp$, 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 $o \vDash c$, 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 $\langle o_1, o_2\rangle \vDash c$, 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 $\langle s, r, a\rangle \vDash \rho$, 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.

4. 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 (Ian et al., 2010) and later extended to ABAC (Xu and Stoller, 2015). 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 (Matthias and Martucci, 2013). 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 WSC(π), 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 $WSC_{cnd}(c)$, is the sum of the WSC of the constituent atomic conditions. The WSC of a constraint c, denoted $WSC_{cns}(c)$, is the sum of the WSC of the constituent atomic constraints. The WSC of a rule is $WSC(\langle st, sc, rt, rc, c, A\rangle) = w_1 WSC_{cnd}(sc) + w_1 WSC_{cnd}(rc) + w_2 WSC_{cns}(c) + w_3|A|$, where $|A|$ is the cardinality of set A, and the w_i are user-specified weights.

5. Greedy algorithm

This section presents our greedy algorithm. It is based on the ABAC policy mining algorithm in Xu and Stoller (2015). The main differences are summarized in Section 2.

Top-level pseudocode appears in Fig. 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. For some functions, we give a description in the text and pseudocode; for others, we give only a description, to save space.

The workset *uncovSP* in Fig. 1 is a priority queue sorted in descending lexicographic order by the quality Q_{sp} of the subject-permission tuple. Informally, $Q_{sp}(\langle s, r, a\rangle)$ 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.

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

The function candidateConstraint(s, r) in Fig. 2 returns a set containing all the atomic constraints that hold between resource r and subject s and satisfy path length constraints described below. It first computes a set cc of candidate constraints using type-correct short paths to each type T reachable from both type(s) and type(r) in graph(CM), which is defined to be a graph with a vertex for each class, and an edge from class c_1 to class 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 case studies. It uses the following auxiliary functions. The function reach(T) returns the set of classes reachable from T in graph(CM), including their superclasses. The function paths(T, T', L) returns all paths from T to T' in graph(CM) whose length is at most L more than the length of the shortest path from T to T'. This reflects our observation that paths in constraints in case studies are the shortest paths between the relevant types or slightly longer. SPED (mnemonic for "subject path extra distance") and RPED (mnemonic for "resource path extra distance") are parameters of the algorithm. We also observe that the subject path and resource path typically do not both have the maximum allowed length in the same constraint, so we introduce a parameter MTPL (mnemonic for "maximum total path length") that limits the sum of the lengths of these paths in a constraint.

The function opFromMul(m, m') returns the relational operator suitable for left and right operands with multiplicity m and m', respectively.

$$\begin{aligned} \text{opFromMul}(m, m') = {} & (\langle m, m'\rangle = \langle \text{many}, \text{many}\rangle \,? \\ & \text{supseteq} : (m = \text{many} \,?\, \text{contains} : \\ & (m' = \text{many} \,?\, \text{in} : \text{equal}))) \end{aligned}$$

The function addCandidateRule(st, s_s, rt, s_r, cc, s_a, *uncovSP*, *Rules*) in Fig. 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 rule set *Rules*. The details of the functions called by addCandidateRule are described next.

The function computeCondition(O, C, L) in Fig. 4 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 $\langle p$, in, $V\rangle$ where V is the collected values of $o.p$ for o in O. A path with multiplicity many may appear in multiple conjuncts, of the form $\langle p$, 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,

```
// Phase 1: Create a set Rules of candidate rules that covers SP_0.
Rules = ∅
// uncovSP contains tuples in SP_0 that are not covered by Rules
uncovSP = SP_0.copy()
while ¬uncovSP.isEmpty()
  // Use highest-quality uncovered tuple as a "seed" for rule creation.
  ⟨s, r, a⟩ = highest-quality tuple in uncovSP according to Q_sp
  cc = candidateConstraint(s, r)
  // s_s contains subjects with permission ⟨r, a⟩ and that have
  // the same candidate constraint for r as s
  s_s = {s' ∈ OM | type(s') = type(s) ∧ ⟨s', r, a⟩ ∈ SP_0
                 ∧ candidateConstraint(s', r) = cc}
  addCandidateRule(type(s), s_s, type(r), {r}, cc, {a}, uncovSP, Rules)
  // s_a is set of actions that s can perform on r
  s_a = {a' | ⟨s, r, a'⟩ ∈ SP_0}
  addCandidateRule(type(s), {s}, type(r), {r}, cc, s_a, uncovSP, Rules)
end while
// Phase 2: Combine rules using least upper bound and inheritance.
// Also, simplify them and remove redundant rules.
mergeRulesAndSimplify(Rules)
mergeRulesInheritance(Rules)
mergeRulesAndSimplify(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 Σ_{ρ∈Rules'} [[ρ]] ⊇ SP_0,
using SP_0 \ [[Rules']] as second argument to Q_rul, and discarding a rule if it does not
cover any tuples in SP_0 currently uncovered by Rules'.
return Rules'
```

Fig. 1 – Greedy algorithm for ReBAC policy mining. Inputs: subject-permission relation SP_0, class model CM, and object model OM. Output: set of rules *Rules'*. The algorithm also has numerical parameters MCSE, MSPL, MRPL, SPED, RPED, and MTPL that limit the considered rules, as described in the text.

```
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 p_1 in paths(type(s), T, SPED)
    for p_2 in paths(type(r), T, RPED) such that |p_1| + |p_2| ≤ MTPL
      cc.add(⟨p_1, opFromMul(multiplicity(p_1), multiplicity(p_2)), p_2⟩)
    end for
  end for
end for
return {c ∈ cc | ⟨s, r⟩ ⊨ c}
```

Fig. 2 – Compute candidate constraints for subject *s* and resource *r*.

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*, denoted valid(ρ), if $[[\rho]] \subseteq SP_0$.

The function generalizeRule(ρ, *cc*, *uncovSP*, *Rules*) in Fig. 5 attempts to generalize rule ρ by adding some 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 tries to generalize ρ using each constraint in *cc* separately, discards the invalid generalizations, sorts the valid generalizations in descending order of the number of covered entitlements in *uncovSP*, recursively tries to further generalize each of them using constraints from *cc* that produced 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 valid, it simply returns ρ. When trying to add an atomic 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 invalid, then there is no valid generalization of ρ using *c*.

```
function addCandidateRule(st, s_s, rt, s_r, cc, s_a, uncovSP, Rules)
// Construct a rule ρ that covers subject-permission tuples {⟨s, r, a⟩ ∈ SP_0 | s ∈ s_s ∧ r ∈ s_r ∧ a ∈ s_a}.
sc = computeCondition(s_s, st, MSPL);
rc = computeCondition(s_r, rt, MRPL)
ρ = ⟨st, sc, rt, rc, ∅, s_a⟩
ρ' = generalizeRule(ρ, cc, uncovSP, Rules)
Rules.add(ρ')
uncovSP.removeAll([[ρ']])
```

Fig. 3 – Compute a candidate rule and add it to *Rules*.

```
function computeCondition(O, C, L)
// First try to characterize set O without using "id" field.
c = new Set()
for each path p s.t. (p is type-correct starting from C) ∧ |p| ≤ L ∧ (p does not contain "id")
  vals = {nav(o, p) | o ∈ O}
  if ⊥ ∉ vals
    if multiplicity(p) ∈ {optional, one}
      c.add(⟨p, in, vals⟩)
    else // multiplicity(p) = many
      I = intersection of the sets in vals
      for val in I
        c.add(⟨p, contains, val⟩)
      end for
    end if
  end if
end for
if [[c]]_C ≠ O
    // "id" field is needed to characterize O.
    c.add(⟨id, in, {nav(o, id | o ∈ O}⟩)
end if
return c
```

Fig. 4 – Compute a condition that characterizes set *O* of objects of type *C*, using paths of length at most *L*.

A *rule quality metric* is a function $Q_{rul}(\rho, SP)$ 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 atomic constraints, and a tertiary preference for rules with shorter paths in atomic constraints, we define

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

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

The preference for more atomic constraints is a heuristic, based on the observation that rules with more atomic constraints tend to be more general than other rules with the same $|[[\rho]] \cap SP|/WSC(\rho)$ (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 pseudocode for generalizeRule in Fig. 5 uses the following auxiliary functions. sPath(*c*) and rPath(*c*) are the subject path and resource path, respectively, used in atomic constraint *c*. rm(*c*, *p*) is the condition obtained by removing the atomic condition on path *p* (if any) from condition *c*. *a*[i..] denotes the suffix of array *a* starting at index i. The loop over i in generalizeRule considers all possibilities for the first atomic constraint in *cc* that gets added to the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in *cc* that get added to the constraint.

The function mergeRulesAndSimplify(*Rules*) repeatedly calls mergeRules and simplifyRules until they have no effect.

The function mergeRules(*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

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

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*. Rule pairs are considered for merging in descending lexicographic order of rule pair quality, where the quality of a rule pair $\langle \rho_1, \rho_2\rangle$ is $\langle \max(q_1, q_2), \min(q_1, q_2)\rangle$ where $q_i = Q_{rul}(\rho_i, SP_0)$. mergeRules(*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

```
function generalizeRule(ρ, cc, uncovSP, Rules)
// split ρ into its components, for convenience.
⟨subType, subCond, resType, resCond, constr, acts⟩ = ρ
// ρbest is highest-quality generalization of ρ
ρbest = ρ
// try to create generalizations of ρ using each constraint in cc. save them in results.
results = new Vector()
for c in cc
  // try to generalize ρ by adding constraint c and eliminating the conjuncts for both
  // paths used in c.
  ρ′ = ⟨subType, rm(subCond, sPath(c)), resType, rm(resCond, rPath(c)), constr ∪ {c}, acts⟩
  // if the paths in c appear in the conditions in ρ and hence have been eliminated in ρ′,
  // and ρ′ is valid, then add ⟨c, ρ′⟩ to results.
  if sPath(c) appears in subCond ∧ rPath(c) appears in resCond ∧ valid(ρ′)
    results.add(⟨c, ρ′⟩)
  else
    // try to generalize ρ by adding constraint c and eliminating the conjunct for the subject path in c.
    ρ′ = ⟨subType, rm(subCond, sPath(c)), resType, resCond, constr ∪ {c}, acts⟩
    if sPath(c) appears in subCond ∧ valid(ρ′)
      results.add(⟨c, ρ′⟩)
    else
      // try to generalize ρ by adding constraint c and eliminating the conjunct for the resource path in c.
      ρ′ = ⟨subType, subCond, resType, rm(resCond, rPath(c)), constr ∪ {c}, acts⟩
      if rPath(c) appears in resCond ∧ valid(ρ′)
        results.add(⟨c, ρ′⟩)
      end if
    end if
  end if
end for
sort results in descending order by Q(⟨c, ρ′⟩) = number of tuples in uncovSP covered by ρ′
cc' = sequence containing the first components of the tuples in results
gen = sequence containing the second components of the tuples in results
for i = 1 to results.length
  // try to further generalize gen[i]
  ρ″ = generalizeRule(gen[i], cc′[i+1 ..], uncovSP, Rules)
    if Qrul(ρ″, uncovSP) > Qrul(ρbest, uncovSP)
    ρbest = ρ″
  end if
end for
return ρbest
```

Fig. 5 – Generalize rule ρ.

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. Removing one conjunct might prevent removal of another conjunct, so it searches for a set of removable 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 $Q_{ac}(\langle p, op, val\rangle) = \langle |val|, |p|, \text{isId}(p), \text{toString(p)}\rangle$, where $|val|$ is 1 if val is atomic and is the cardinality of val is a set, and 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 tuples 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. (5) If the subject condition contains an atomic condition of the form $p = c$, and the constraint contains an atomic constraint of the form $p = p'$, then replace that atomic constraint with the atomic condition $p' = c$ in the resource condition (note that this is a form of constant propagation); and similarly for the symmetric situation in which the resource condition contains such an atomic condition, etc. (6) Remove cycles in the paths in the conditions and constraint, if the resulting rule is valid and the resulting policy still covers all of SP_0. A cycle is a path that navigates from some class C back to class C. For example, for the class model in Fig. 6, the path "physician.consultations.patient" contains the cycle "physician.consultations", which navigates from Consultation back to Consultation.

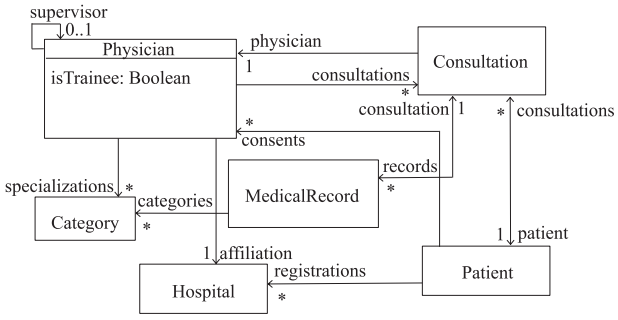

Fig. 6 – Class model for Electronic Medical Records (EMR) policy.

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 valid, if any. 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'.

Optimization. Our implementation includes two optimizations not reflected in the pseudocode. (1) Meanings of atomic conditions, conjunctions of atomic conditions, atomic constraints, conjunctions of atomic constraints, and rules are cached are re-used. (2) The first loop in Fig. 1 processes seed tuples in batches of 1000, and calls mergeRules on the rules generated from each batch of seed tuples before adding the resulting rules to *Rules*. This reduces the size of *Rules* at the end of that loop. This reduces the overall running time, because mergeRules is quadratic in the number of rules.

5.1. *Example*

We illustrate the algorithm on a fragment our *Electronic Medical Record (EMR) sample policy*, a ReBAC policy based on the EBAC policy in Jasper et al. (2015). It controls access by physicians and patients to electronic medical records, based on institutional affiliations, patient-physician consultations (each EMR is associated with a consultation), supervisor relationships among physicians, etc. The class model is in Fig. 6. We developed a pseudorandom algorithm that creates object models of varying size for this policy; the algorithm has a size parameter N, and the numbers of physicians, patients, consultations, medical records, and hospitals are proportional to N. We generated a object model with $N = 15$ for this example. When describing the execution of the algorithm, we refer to objects by the value of the "id" field. We use id's such as phy0, phy1, ... for instances of Physician; consult0, consult1, ... for instances of Consultation; and so on.

The policy contains 6 rules. To keep this example small, we consider here only one rule, namely, the rule in Eq. (1).

Our algorithm selects subject-permission tuple ⟨phy0, consult8, createMedicalRecord⟩ as the first seed, and then calls candidateConstraint to compute the set *cc* of atomic constraints that hold between phy0 and consult8. *cc* includes c_1, c_2, and c_3 where

c_1 = subject = resource.physician

c_2 = subject.affiliation ∈ resource.patient.registrations

c_3 = subject.affiliation ∈ resource.patient.consents.affiliation.

The first call to addCandidateRule calls computeCondition to compute a condition *sc* that characterizes the set of subjects with permission ⟨createMedicalRecord, consult8⟩ and the same candidate constraint as phy0 for consult8. Condition *sc* contains the conjunct subject.isTrainee = false along with conjuncts, such as subject.consultations.physician.id = phy0, later removed by simplification. The second call to computeCondition returns a condition *rc* that characterizes {consult8}; it contains conjuncts such as resource.physician.id = phy0, which is later removed by generalizeRule, and resource.physician.affiliation.id = hosp1, which is later removed by simplification.

addCandidateRule creates rule ρ_1 = ⟨Physician, *sc*, MedicalRecord, *rc*, ∅, {createMedicalRecord}⟩ and then calls generalizeRule, which sorts *cc* and then attempts to add the atomic constraints in it to ρ_1, removing conjuncts for some of the paths used in them. In one of the recursive calls, the current rule ρ_2 contains c_1 and c_2. Adding c_3 (which uses the same subject path as c_2) to this rule worsens rule quality, so generalizeRule returns ρ_2. To see the importance of sorting *cc*, note that, if the algorithm had added c_1 and c_3 before trying to add c_2, then generalizeRule would return a rule containing c_1 and c_3, worsening overall policy quality. The second call to addCandidateRule generates the same candidate rule as the first call, because createMedicalRecord is the only action phy0 can perform on consult8. The algorithm generates other candidate rules from other seeds, and then calls mergeRules, which merges ρ_1 with rules created from permissions of other physicians to create medical records associated with other consultations. The merged rule is simplified by simplifyRules to produce the desired rule ρ.

6. Evolutionary algorithm

Our grammatical evolution algorithm uses the Context-Free Grammar Genetic Programming (CFGGP) approach, in which individuals (which in our context are ReBAC rules) are represented directly as derivation trees of a context-free grammar (CFG). This is simpler than alternative approaches in which derivation trees are encoded as, e.g., binary strings.

Classical CFGGP uses two genetic operators to evolve derivation trees: (1) 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 (2) 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.

Our algorithm uses these two classical CFGGP genetic operators (with slight variations, to reflect the focus of the

evolutionary search on rules that cover a given seed tuple). However, we found that the algorithm, with these genetic operators alone, gave poor results, because some mutations that are especially useful in our setting had very low probability. We solved this problem by introducing additional mutation operators. For example, we introduced 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, which changes at least one condition and the constraint. The same effect can be achieved by two separate mutations to the same rule, but the probability of achieving it that way is much lower. Although these new mutation operators are specialized to ReBAC policy mining, they are independent of details of our policy language and hence should be equally applicable and effective to extensions of the policy language with additional primitive datatypes (numbers, sequences, etc.) and relational operators (numeric inequality, prefix-of, etc.).

As sketched in Section 1, our evolutionary algorithm has two phases: phase 1 constructs a candidate policy, and phase 2 tries to improve the candidate rules by further mutating them. The improvement phase might seem redundant, because it uses essentially the same mutations as the first phase. The key difference is that, in phase 1, the benefit of a mutation is evaluated by its effect on rule quality, and in phase 2, it is evaluated in the context of the entire candidate policy by its effect on policy quality. For example, consider a mutation that transforms a candidate rule ρ into ρ', such that ρ' covers fewer subject-permission tuples, has lower WSC, and has lower rule quality. If this mutation occurs in phase 1, ρ' might survive, but it is likely to be discarded, due to its lower rule quality. If this mutation occurs in phase 2, and if the tuples covered by ρ and not by ρ' are also covered by other rules in the candidate policy, ρ' will definitely replace ρ in the candidate policy, because this change reduces the policy's WSC and does not change the policy's meaning.

Grammar generation is performed before the main part of the evolutionary algorithm, to specialize the generic grammar of ORAL to a specific input. The language of the generated grammar contains rules satisfying the restrictions: (1) constants are limited to those appearing in the object model, (2) class names and field names are limited to those appearing in the class model, (3) paths in conditions and constraints are type-correct, based on the class model, and satisfy the same length limits as in the greedy algorithm, and (4) actions are limited to those appearing in the given subject-permission relation. The grammar generation algorithm pre-computes all atomic conditions and atomic constraints satisfying these restrictions. For a type t, let $\{c_{t,1}, c_{t,2}, \ldots, c_{t,n_t}\}$ denote the set of atomic conditions on objects of type t that satisfy these restrictions.

The starting non-terminal N_{rule} has alternatives corresponding to rules with different subject and resource types. Specifically, each alternative for N_{rule} is a rule with two non-terminals: a non-terminal $N_{rule(t_1,t_2)}$ that generates all components of a rule with subject type t_1 and resource type t_2 except for the action set component (which is independent of the types), and a non-terminal N_{act} that generates subsets of the actions that appear in SP_0. Each alternative for $N_{rule(t_1,t_2)}$ contains non-terminals $N_{cond(t_1)}$ and $N_{cond(t_2)}$ to generate the subject condition and resource condition, respectively, and a non-terminal $N_{cons(t_1,t_2)}$ to generate the constraint. The non-terminal $N_{cond(t)}$ generates conditions on objects of type t. Specifically, it generates a sequence of non-terminals $N_{t,1}, N_{t,2}, \ldots, N_{t,n_t}$, where each $N_{t,i}$ can generate either atomic condition $c_{t,i}$ or the empty string; this allows the condition to contain an arbitrary subset of the atomic conditions on objects of type t. The non-terminal $N_{cons(t_1,t_2)}$ generates constraints relating objects of types t_1 and t_2; the productions for it are defined in a similar way as the productions for conditions.

Pseudocode for the main part of our evolutionary algorithm appears in Fig. 7. All random choices follow a uniform distribution, unless a different probability distribution is specified. In our experiments, values of the numerical parameters are: *popSize*=200, *nGenerationsSearch*=2000, *nTournament*=15, *nGenerationsImprove* = 1000. These values, and other numerical parameter values mentioned below, were selected based on tuning experiments, described in Section 8.

Function initialPopulation(⟨*s, r, a*⟩, *Rules, uncovSP*) creates an initial population for the evolutionary search for a high-quality rule that covers the seed ⟨*s, r, a*⟩ and other tuples. It is implicitly parameterized by the desired population size *popSize*. Half of the desired rules are generated using method 1; the other half are generated using method 2.

Method 1 (candidate rules generated as in greedy algorithm plus random variants): Generate candidate rules from *seed, uncovSP*, and *Rules* in the same way as the two calls to addCandidateRule in Fig. 1 and add them to the initial population; this ensures that it contains at least 1 valid rule that covers at least one uncovered tuple. To generate the remaining rules, repeatedly randomly select a rule currently in the initial population, remove randomly selected atomic conditions from the subject condition until the number of remaining atomic conditions equals a target number randomly selected in the interval 1..7, do the same for the resource condition, remove randomly selected atomic constraints until the number of remaining atomic constraints equals a target number randomly selected in the interval 1..3, and add the resulting rule to the initial population.

Method 2 (random candidate rules): Each rule has subject type type(*s*) or one of its ancestors, resource type type(*r*) or one of its ancestors, subject condition and resource condition selected as described below, randomly selected constraint consistent with the selected subject type and resource type, and action set {*a*}. If type(*s*) has an ancestor, then the probability of using type(*s*) as the subject type is 0.8, and the probability of using an ancestor (selected uniformly at random among the ancestors) is 0.2; similarly for the resource type. To generate the subject condition and resource condition, we randomly select among the following three cases, and then select randomly select a condition within the selected case: no condition (i.e., the condition is always true), a condition on a single-valued path, and an arbitrary condition.

Rule quality is measured using the same fitness function f as Medvet et al. (2015) (our definition is slightly simplified but equivalent): $f(\rho) = \langle \mathrm{FAR}(\rho), \mathrm{FRR}(\rho), \mathrm{ID}(\rho), \mathrm{WSC}(\rho) \rangle$, where the *false acceptance rate* is $\mathrm{FAR}(\rho) = |[\![\rho]\!] \setminus uncovSP|$, the *false rejection rate* is $\mathrm{FRR}(\rho) = |uncovSP \setminus [\![\rho]\!]|$, and $\mathrm{ID}(\rho)$ equals 2 if the subject condition and resource condition both contain an atomic

```
// Phase 1: Construct candidate policy, using evolutionary search to find one rule at a time.
Rules = ∅
uncovSP = SP_0.copy()
while ¬uncovSP.isEmpty()
  ⟨s, r, a⟩ = highest-quality tuple in uncovSP using Q_sp metric      // seed for this iteration
  pop = initialPopulation(⟨s, r, a⟩, Rules, uncovSP)
  for gen = 1 to nGenerationsSearch
    op = an operator selected from searchOps using probability distribution searchOpDist
    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
  ρ = the highest-quality rule in pop
  if valid(ρ)
    Rules.add(ρ)
    uncovSP.removeAll([[ρ]])
  end if
end while
// Phase 2: Improve the candidate rules by further mutating them.
for each ρ in Rules
  for gen = 1 to nGenerationsImprove
    if gen = nGenerationsImprove/2 ∧ (all attempted improvements to ρ failed)
      break
    end if
    op = an operator selected from improveOps using probability distribution improveOpDist
    ρ' = the rule generated by applying op to ρ
    if wellFormed(ρ') ∧ valid(ρ') ∧ ID(ρ') ≤ ID(ρ)
      redundant = {ρ_0 ∈ Rules | [[ρ_0]] ⊆ [[ρ']]}
      if (Rules ∪ {ρ'} \ redundant) covers SP_0 and has lower WSC than Rules
        Rules.removeAll(redundant)
        Rules.add(ρ')
      end if
    end if
  end for
end for
mergeRulesAndSimplify2(Rules)
return Rules
```

Fig. 7 – Evolutionary algorithm for ReBAC policy mining. Inputs: subject-permission relation SP_0, class model CM, and object model OM. Output: set of rules $Rules$. The algorithm also has numerical parameters that determine population size, number of generations, etc., as described in the text.

condition with path "id", equals 1 if exactly one of them does, and equals 0 if neither of them does. The fitness ordering is lexicographic order on these tuples, where smaller is better.

The set of genetic operators used in the search phase, denoted *searchOps*, contains: (1) single mutation: first, randomly select whether to mutate the subject condition, resource condition, or constraint, then randomly select a non-terminal N in that part of the derivation tree, and then randomly re-generate the subtree rooted at N; (2) double mutation: same as single mutation, except, in the first step, choose two out of the three possibilities, and then perform the remaining steps for both of them; (3) action mutation: in the action set component of the rule, randomly add or remove actions that subject s can perform on r according to SP_0, subject to the restriction that we never remove action a, where $\langle s, r, a\rangle$ is the seed tuple for this search; (4) simplify mutation: remove one randomly selected atomic condition (from the subject condition or resource condition) or atomic constraint; (5) crossover: randomly select a non-terminal N in the subtree for the subject condition, resource condition, or constraint in one parent, find the same non-terminal in the other parent (if it does not appear, select a different non-terminal in the first parent), and swap the subtrees rooted at those two occurrences of N.

We describe the genetic operators as if they directly manipulate abstract syntax trees, because this allows a higher-level and more intuitive presentation. However, the genetic operators actually manipulate derivation trees of the generated grammar.

searchOpDist specifies the probability of selecting each genetic operator in *searchOps*. First, it selects the type of genetic

Table 1 – Policy sizes. #cond/rule and #constr/rule are the average numbers of conditions per rule and constraints per rule, respectively. For the given value of N, #obj is the average number of objects in the object model, and #field/obj is the average number of fields (including "id" field) per object in the object model. Averages are over 30 pseudorandom object models for each policy.

Policy	#rules	#cond/rule	#constr/rule	#classes	N	#obj	#field/obj	$\|SP_0\|$
EMR	6	0.17	1.3	6	15	344	3.5	708
healthcare	9	0	1.1	12	5	737	3.5	2207
project mgmt.	13	0.08	1.2	15	5	181	2.7	322
university	10	0.40	0.70	10	5	731	2.2	2439
e-document	39	2.3	0.59	16	125	421	5.9	2687
workforce mgmt.	27	1.7	0.63	29	10	411	3.7	1739

operator, selecting mutation with probability 0.9, or crossover with probability 0.1. If mutation is selected, the probability of selecting each of the four types of mutation is proportional to its weight, where single mutation, action mutation, and simplify mutation each have weight 1, and double mutation has weight 0.7.

The set of genetic operators used in the improvement phase, denoted *improveOps*, contains: (1) single mutation; (2) double mutation; (3) type+single mutation: randomly select whether to replace the subject type, resource type, or both with their parent types (if those parents exist), apply a single mutation, check whether the resulting rule is well-formed (because the unchanged condition or constraint might be inconsistent with the changed type), and if not, discard it; (4) type+double mutation: same as type+single mutation, except with a double mutation instead of a single mutation.

improveOpDist specifies the probability of selecting each genetic operator in *improveOps*. The probabilities are: single mutation, 0.09; double mutation, 0.81; type+single mutation, 0.01; type+double mutation, 0.09.

Function mergeRulesAndSimplify2 is the same as mergeRulesAndSimplify except that it incorporates one additional simplification: replace the subject type or resource type with a child of that type, if the policy still covers SP_0.

7. Sample policies and case studies

We developed four sample policies, which have non-trivial and realistic rules, but are relatively small. We also translated two large case studies into ORAL. They were developed by Decat et al. based on the access control requirements for Software-as-a-Service (SaaS) applications offered by real companies (Decat et al., 2014a,b). We translate their 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.

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 (e.g., normal, uniform, and Zipf distributions). The object model generation algorithm for each policy is parameterized by a size parameter N; for most classes, the number of instances is selected from a normal distribution whose mean is linear in N. Table 1 shows several metrics of the size of the rules, class model, and object model in each policy.

The *Electronic Medical Record (EMR) sample policy* is described in Section 5.1.

The *healthcare sample policy*, based on the ABAC policy in Xu and Stoller (2015), 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.

The *project management sample policy*, based on the ABAC policy in Xu and Stoller (2015), 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.

The *university sample policy*, based on the ABAC policy in Xu and Stoller (2015), 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.

Rewriting the preceding three policies in ReBAC allows numerous aspects to be expressed more naturally than in ABAC. This is reflected in rules that use paths with length greater than one, not counting occurrences of "id". For example, consider the constraint "subject.teams contains resource.record.patient.treatingTeam" in the above example rule from the healthcare policy. In the ReBAC policy, "treatingTeam" is, naturally, an attribute of Patient. In the original ABAC policy, there is no way to navigate from the HR item to the patient; to circumvent this limitation, a patient's "treatingTeam" attribute is (unnaturally) duplicated in each HR item in each HR for that patient.

The *e-document case study*, based on Decat et al. (2014a), 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 and mailing them). 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. One sample tenant is a large bank, which controls permissions to send and read

documents based on (1) employee attributes such as department and projects, (2) document attributes such as document type, related project (if any), and presence of confidential or personal information, and (3) the bank customer to which the document is being sent. Some tenants have semi-autonomous sub-organizations, modeled as sub-tenants, each with its own specialized policy rules. 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 Decat et al. (2014b), is for a SaaS workforce management application provided by a company, pseudonymously called eWorkforce, that 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, its workforce suppliers, or subcontractors of its workforce suppliers receive work orders to work on those tasks, and appointments are scheduled if appropriate. Warehouse operators receive requests for required supplies. The overall policy contains rules governing the employees of eWorkforce, as well as specific policies for some sample tenants, including PowerProtection (a provider of power protection equipment and installation and maintenance services) and TelCo (a telecommunications provider, including installation and repair services). Permissions to view, assign, and complete tasks are based on each subject's position, the assignment of tasks to technicians, the set of technicians each manager supervises, the contract (between eWorkforce and a tenant) that each work order is associated with, the assignment of contracts to departments within eWorkforce, etc. The numbers of helpdesk suppliers, workforce providers, subcontractors, 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. The parameter values are similar across policies, though they vary by 1 or 2. A reasonable parameter value selection strategy is to start with values similar to these, perhaps on the lower end, and increase them slightly if the mined policy is unsatisfactory.

8. Evaluation

To evaluate the effectiveness of our algorithms, we start with a ReBAC policy, generate ACLs representing the subject-permission relation, run our algorithms on the ACLs along with the class model and object model, and compare the mined ReBAC policy with the policy produced by applying simplifyRules to the original policy; we refer to the latter as the *simplified original policy*. If the mined policy is similar to the simplified original policy, the policy mining algorithm succeeded in discovering the rules that are implicit in the ACLs. Comparison with the simplified original policy is a more robust measure of the algorithm's ability to discover high-level rules than comparison with the original policy, because the original policy is not always the simplest. For our four sample policies, the simplified original policy is identical to the original policy; for the two large case studies, the simplified original policy has lower WSC than the original policy.

Our algorithm is implemented in Java. Experiments were run using Java 8 on Windows 10 on an Intel i7-6770HQ CPU. The code and data are available at http://www.cs.stonybrook.edu/~stoller/software/.

8.1. 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 types, 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 the average of $J(\{st_1\}, \{st_2\})$, $J(sc_1, sc_2)$, $J(\{rt_1\}, \{rt_2\})$, $J(rc_1, rc_2)$, $J(c_1, c_2)$ and $J(A_1, A_2)$. The *syntactic similarity of rule sets* $Rules_1$ and $Rules_2$, $SynSim(Rules_1, Rules_2)$, is the average, over rules ρ in $Rules_1$, of the syntactic similarity between ρ and the most similar rule in $Rules_2$.

Semantic similarity. Semantic similarity measures the similarity of the meanings of rules. The *semantic similarity of rules* ρ_1 and ρ_2 is $J([[\rho_1]], [[\rho_2]])$. We extend this to *rule-wise semantic similarity of policies* $RSemSim(Rules_1, Rules_2)$ 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.* This metric is slightly more abstract than syntactic similarity, because it ignores syntactic differences that do not affect the meaning of a rule, such as including a conjunct that is unnecessary because it is implied by another conjunct or a constraint.

Parameter tuning. Our evolutionary algorithm has several parameters (population size, number of generations, etc.). The parameters and their values used in our experiments are presented in Section 6. We determined those values through a series of experiments, in which we started with initial guesses at good values of the parameters, varied one parameter, selected the value that gave the best average policy similarity results, and then proceeded to vary the next parameter (in a somewhat arbitrary order, except that we vary parameters used in initialization before parameters used in phase 1 before parameters used in phase 2). After varying each parameter once, we varied some parameters again, and found little change in the results, so we did not bother with an exhaustive optimization process that would consider all combinations of values of all parameters.

8.2. Policy similarity results

Table 2 shows the results of policy similarity measurements. The policy size parameter N has the values shown in Table 1. We set all weights w_i in the definition of WSC to 1.

Table 2 – Policy similarity results. Evol. and Greedy refer to the rules mined by the evolutionary algorithm and greedy algorithm, respectively. SimpOrig refers to the simplified original rules. When computing policy similarity, the first argument to *SynSim* and *RSemSim* is the mined rules, and the second argument is the simplified original rules. μ is the mean over 30 pseudorandom object models, and σ is the standard deviation. Similarity results for the evolutionary algorithm are emphasized with bold font, since they are as good as or better than the results for the greedy algorithm in all cases.

Policy	Syntactic similarity				Rule-wise semantic sim.				WSC		
	Evol.		Greedy		Evol.		Greedy		SimpOrig	Evol.	Greedy
	μ	σ	μ	σ	μ	σ	μ	σ	μ	μ	μ
EMR	**0.99**	0.01	0.99	0.01	**1.00**	0.00	1.00	0.00	49	49	50
healthcare	**1.00**	0.00	1.00	0.00	**1.00**	0.00	1.00	0.00	54	54	54
project mgmt.	**1.00**	0.00	1.00	0.00	**1.00**	0.00	1.00	0.00	76	76	76
university	**1.00**	0.00	1.00	0.00	**1.00**	0.00	1.00	0.00	54	54	54
e-document	**0.90**	0.03	0.86	0.02	**0.86**	0.08	0.72	0.07	250	326	416
workforce mgmt.	**0.92**	0.02	0.81	0.02	**0.96**	0.02	0.90	0.03	208	185	229

For the four sample policies, for both of our policy mining algorithms, the mined policy is identical to the simplified original policy, except for one minor syntactic variation in one conjunct of one condition of one rule of the EMR policy (the variant is semantically equivalent to the original conjunct; this is reflected in the perfect rule-wise semantic similarity). For the two large case studies, the evolutionary algorithm does better than the greedy algorithm: for e-document, the syntactic similarity and rule-wise semantic similarity are 4% and 14% higher, respectively (but the running time is longer, as discussed below); for workforce management, they are 11% and 6% higher, respectively.

The *e-document case study* is the most difficult for our algorithms. Both algorithms do well on 37 of the 39 input rules. The greedy algorithm fails to discover the conditions and constraints needed for the remaining two rules, producing instead rules that identify several employees individually by enumerating their id's. One of these rules is challenging because its subject condition, resource condition, and constraint are each non-trivial, and the resource condition (resource.type.id in {Invoice,SalesOffer,Contract}) involves a three-element set. The evolutionary algorithm does better on these rules, discovering them in most but not all object models. The significantly higher WSC of the mined policies produced by both algorithms, relative to the simplified original rules, is due to their difficulty with these two rules. The relatively large gap between the syntactic similarity and rule-wise semantic similarity for the greedy algorithm on this policy is also due to its difficulty with the aforementioned rule, and reflects the fact that relatively small syntactic differences (e.g., changing one constant in one condition) can cause a relatively large change in the meaning of a rule.

For the *workforce management case study*, the evolutionary algorithm produces policies that have even lower WSC (about 11% lower) than the simplified original policy, and hence do not have perfect similarity. The greedy algorithm produces policies with somewhat higher WSC (about 10% higher) than the simplified original policy. The relatively large gap between the syntactic similarity and rule-wise semantic similarity for the greedy algorithm on this policy is due to the heuristic that prefers constraints over conditions. This heuristic typically helps increase the generality of rules, but is not helpful for some rules in this policy. The evolutionary algorithm does not incorporate this heuristic and achieves higher syntactic similarity.

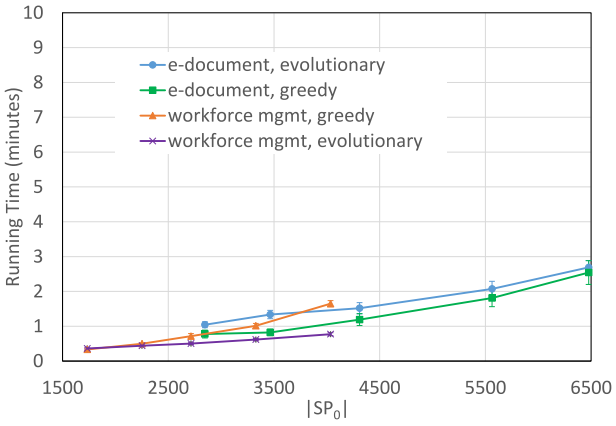

Fig. 8 – Running time of both algorithms on the case studies, as a function of the number of subject-permission tuples.

To evaluate the benefit of our specialized genetic operators, we ran a variant of our algorithm modified to use only the two classic genetic operators (single mutation and crossover), on the two case studies, using the same 30 pseudo-random object models for each policy. Eliminating the specialized genetic operators reduces the average policy similarity slightly for the e-document case study (from 0.90 to 0.89 for syntactic similarity, and from 0.86 to 0.84 for rule-wise semantic similarity) and significantly for the workforce management case study (from 0.92 to 0.82 for syntactic similarity, and from 0.96 to 0.79 for rule-wise semantic similarity).

8.3. *Performance results*

Fig. 8 shows the running time of both algorithms on the case studies as a function of ACL policy size $|SP_0|$. Each data point is the average over 10 pseudo-random object models. Error bars (too small to see in some cases) show 95% confidence intervals using Student's t-distribution. We see that the

algorithms have similar performance on both case studies. The slopes of the best-fit lines on a log–log plot of the data are: for e-document, 1.5 for the greedy algorithm, and 1.1 for the evolutionary algorithm; for workforce management, 1.9 for the greedy algorithm, and 0.9 for the evolutionary algorithm. This is an encouraging indicator of the algorithms' scalability: they can mine dozens of complex rules from ACLs with several thousand entries in minutes, and the growth in running time, as a function of the number of ACL entries, is less than quadratic for the greedy algorithm, and is close to linear for the evolutionary algorithm.

9. Conclusions and future work

A long-standing trend in research on access control policy mining is to handle increasingly expressive policy languages, starting with flat RBAC (Martin et al., 2003), advancing to RBAC with role hierarchy (Jürgen and Ulrike, 2005), followed by RBAC with extensions such as temporal constraints (Barsha et al., 2013) and parameterized roles (Xu and Stoller, 2013), and then ABAC (Xu and Stoller, 2015). This paper continues that trend. We introduced ORAL, an ReBAC policy language formulated as an object-oriented extension of ABAC, defined the ReBAC policy mining problem, and presented the first two algorithms for that problem. Our evaluation on four sample policies and two larger and more complex case studies, based on SaaS applications offered by real companies, demonstrate the effectiveness of our algorithms.

There are many interesting directions for future work on access control policy mining. One obvious direction is policy mining for ReBAC languages with additional features: additional data types and corresponding relational operators (e.g., integers with inequalities), negation, temporal or spatial constraints, actions involving multiple resources, etc. Another practical direction for future work is mining ReBAC policies from incomplete data (e.g., access logs instead of ACLs) or noisy data (e.g., extraneous permissions, or incorrect attribute values). Yet another direction is to explore incremental approaches to policy mining to support policy evolution.

Acknowledgments

We thank Eric Medvet for helpful discussions about grammatical evolution.

REFERENCES

Alberto B, Andrea De L, Eric M, Fabiano T. Learning text patterns using separate-and-conquer genetic programming. In: Proceedings of the 18th European conference on genetic programming (EuroGP). Springer; 2015. p. 16–27.

Alessandro C, Roberto DP, Alberto O, Nino Vincenzo V. A formal framework to elicit roles with business meaning in RBAC systems. In: Proceedings of the 14th ACM symposium on access control models and technologies (SACMAT); 2009. p. 85–94.

Alessandro C, Roberto DP, Nino Vincenzo V. A business-driven decomposition methodology for role mining. Comput Secur 2012;31(7):844–55.

Barsha M, Shamik S, Jaideep V, Vijayalakshmi A. Mining temporal roles using many-valued concepts. Comput Secur 2016a;60:79–94.

Barsha M, Shamik S, Jaideep V, Vijayalakshmi A. A survey of role mining. ACM Comput Surv 2016b;48(4) 50:1–50:37.

Barsha M, Shamik S, Vijayalakshmi A, Jaideep V. Toward mining of temporal roles. In: Proceedings of the 27th annual IFIP WG 11.3 conference on data and applications security and privacy (DBSec). Springer; 2013. p. 65–80. 7964 of *Lecture Notes in Computer Science*.

Carlos C, Thilo W, David B. Mining abac policies from sparse logs. In: Proceedings of the 3rd IEEE European symposium on security and privacy (EuroS&P 2018). ACM; 2015. p. 2141–8.

Carminati B, Ferrari E, Perego A. Enforcing access control in web-based social networks. ACM Trans Inf Syst Secur 2009;13(1):1–38.

Federal Chief Information Officer Council. Federal Identity Credential and Access Management (FICAM) roadmap and implementation guidance, version 2.0. 2011. http://www.idmanagement.gov/documents/ficam-roadmap-and-implementation-guidance.

David F, Ramaswamy C, Vincent H, Rick K. A comparison of Attribute Based Access Control (ABAC) standards for data service applications: Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC). National Institute of Standards and Technology; 2016. NIST Special Publication 800–178.

Medvet E, Alberto B, Barbara C, Elena F. Evolutionary inference of attribute-based access control policies. In: Proceedings of the 8th international conference on evolutionary multi-criterion optimization (EMO). Part I. Springer; 2015. p. 351–65. volume 9018 of *Lecture Notes in Computer Science*.

Gates CE. Access control requirements for Web 2.0 security and privacy. Proceedings of the IEEE web 2.0 security & privacy workshop (W2SP 2007), 2007.

Glenn B, Michael H, Philip F, Ida S. Relationship-based access control: Its expression and enforcement through hybrid logic. In: Proceedings of the second ACM conference on data and application security and privacy (CODASPY). ACM; 2012. p. 117–24.

Haibing L, Jaideep V, Vijayalakshmi A. Optimal boolean matrix decomposition: application to role engineering. In: Proceedings of the 24th international conference on data engineering (ICDE). IEEE; 2008. p. 297–306.

Hu H, Ahn G-J, Jorgensen J. Multiparty access control for online social networks: model and mechanisms. IEEE Trans Knowl Data Eng 2013;25(7):1614–27.

Hu VC, David F, Rick K, Friedman AR, Lang AJ, Cogdell MM, Adam S, Kenneth S, Robert M, Karen S. Guide to Attribute Based Access Control (ABAC) definition and considerations (final draft). National Institute of Standards and Technology; 2013. NIST Special Publication 800–162.

Ian M, Hong C, Tiancheng L, Qihua W, Ninghui L, Elisa B, Calo SB, Jorge L. Mining roles with multiple objectives. ACM Trans Inf Syst Secur 2010;13(4) 36:1–36:35.

Jaideep V, Vijayalakshmi A, Janice W. RoleMiner: mining roles using subset enumeration. In: Proceedings of the 13th ACM conference on computer and communications security (CCS). ACM; 2006. p. 144–53.

Jaideep V, Vijayalakshmi A, Janice W, Qi G. Role engineering via prioritized subset enumeration. IEEE Trans Dependable Secur Comput 2010;7(3):300–14.

Jaideep V, Vijayalakshmi A, Qi G. The role mining problem: finding a minimal descriptive set of roles. In: Proceedings of the 12th ACM symposium on access control models and technologies (SACMAT). ACM; 2007. p. 175–84.

Crampton J, Sellwood J. Path conditions and principal matching: a new approach to access control. In: Proceedings of the 19th

ACM symposium on access control models and technologies (SACMAT). ACM; 2014. p. 187–98.
Jasper B, Maarten D, Bert L, Wouter J. Entity-based access control: supporting more expressive access control policies. In: Proceedings of the 31st annual computer security applications conference (ACSAC 2015). ACM; 2015. p. 291–300.
Jürgen S, Ulrike S. Role mining with ORCA. In: Proceedings of the 10th ACM symposium on access control models and technologies. ACM; 2005. p. 168–76.
Ludwig F, Günther P. HyDRo – hybrid development of roles. In: Proceedings of the 4th international conference on information systems security (ICISS). Springer; 2008. p. 287–302.
Decat M, Jasper B, Bert L, Wouter J. The e-document case study: functional analysis and access control requirements. Department of Computer Science, KU Leuven, 2014a. CW Reports CW654.
Decat M, Jasper B, Bert L, Wouter J. The workforce management case study: functional analysis and access control requirements. Department of Computer Science, KU Leuven, 2014b. CW Reports CW655.
Mario F, Buhmann JM, David A B. Role mining with probabilistic models. ACM Trans Inf Syst Secur 2013;15(4):1–28.
Martin K, Dalia S, Gerhard S. Role mining—revealing business roles for security administration using data mining technology. In: Proceedings of the eighth ACM symposium on access control models and technologies (SACMAT). ACM; 2003. p. 179–86.
Matthias B, Martucci LA. Formal definitions for usable access control rule sets—from goals to metrics. Proceedings of the ninth symposium on usable privacy and security (SOUPS). ACM, 2013.
McKay RI, Nguyen Xuan H, Peter Alexander W, Yin S, Michael O. Grammar-based genetic programming: a survey. Genet Program Evolvable Mach 2010;11(3):365–96.
Nino Vincenzo V, Jaideep V, Vijay A, Alessandro C. Role engineering: from theory to practice. In: Proceedings of the second ACM conference on data and application security and privacy (CODASPY). ACM; 2012. p. 181–92.
Fong PWL. Relationship-based access control: protection model and policy language. In: Proceedings of the first ACM conference on data and application security and privacy (CODASPY). ACM; 2011. p. 191–202.
Qi G, Jaiddeep V, Vijayalakshmi A. The role hierarchy mining problem: discovery of optimal role hierarchies. In: Proceedings of the 2008 annual computer security applications conference (ACSAC). IEEE Computer Society; 2008. p. 237–46.
Safaá H, Nora CB, Frédéric C. Role mining to assist authorization governance: how far have we gone? Int J Secure Softw Eng 2012;3(4):45–64.
Stoller SD, Thang B. Mining hierarchical temporal roles with multiple metrics. J Comput Secur 2017;26(1):121–42.
Thang B, Stoller SD, Jiajie L. Mining relationship-based access control policies. Proceedings of the 22nd ACM symposium on access control models and technologies (SACMAT 2017). ACM Press, 2017.
Whigham PA. Grammatically-based genetic programming. In: Rosca JP, editor. In: Proceedings of the workshop on genetic programming: from theory to real-world applications; 1995. p. 33–41. Tahoe City, California, USA.
Cheng Y, Park J, Sandhu RS. A user-to-user relationship-based access control model for online social networks. In: Proceedings of the 26th annual IFIP WG 11.3 conference on data and applications security and privacy (DBSec). Springer; 2012. p. 8–24. volume 7371 of *Lecture Notes in Computer Science*.
Xu Z, Stoller SD. Algorithms for mining meaningful roles. In: Proceedings of the 17th ACM symposium on access control models and technologies (SACMAT). ACM; 2012. p. 57–66.
Xu Z, Stoller SD. Mining parameterized role-based policies. Proceedings of the third ACM conference on data and application security and privacy (CODASPY). ACM, 2013.
Xu Z, Stoller SD. 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). Springer-Verlag; 2014a. p. 276–91. volume 8566 of *Lecture Notes in Computer Science*.
Xu Z, Stoller SD. Mining attribute-based access control policies. IEEE Trans Dependable Secure Comput 2015;12(5):533–45.
Xu Z, Stoller SD. Mining attribute-based access control policies from role-based policies. Proceedings of the 10th international conference & expo on emerging technologies for a smarter world (CEWIT 2013). IEEE Press, 2014b.

Thang Bui is a doctoral student in the Computer Science Department at Stony Brook University. He received his Bachelor's degree in Computer Science from Stony Brook University in 2016. His primary research area is computer security, focusing on access control.

Scott D. Stoller is a Professor in the Computer Science Department at Stony Brook University. He received his Bachelor's degree in Physics, summa cum laude, from Princeton University in 1990 and his Ph.D. degree in Computer Science from Cornell University in 1997. He is the author or co-author of over 110 refereed research publications that have been cited over 4100 times, according to Google Scholar.

Jiajie Li his Bachelor's degree in Applied Mathematics & Statistics and the Honors Program in Computer Science in 2017, and his Master's degree in Computer Science in 2018, both from Stony Brook University. His primary research area is computer security, focusing on access control.