A Decision Tree Learning Approach for Mining Relationship-Based Access Control Policies

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

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

ABSTRACT

Relationship-based access control (ReBAC) provides a high level of expressiveness and flexibility that promotes security and information sharing, by allowing policies to be expressed in terms of chains of relationships between entities. ReBAC policy mining algorithms have the potential to significantly reduce the cost of migration from legacy access control systems to ReBAC, by partially automating the development of a ReBAC policy.

This paper presents new algorithms, called DTRM (Decision Tree ReBAC Miner) and DTRM$^-$, based on decision trees, for mining ReBAC policies from access control lists (ACLs) and information about entities. Compared to state-of-the-art ReBAC mining algorithms, our algorithms are significantly faster, achieve comparable policy quality, and can mine policies in a richer language.

CCS CONCEPTS

• **Security and privacy → Access control**.

KEYWORDS

security policy mining; attribute-based access control; relationship-based access control; decision trees

ACM Reference Format:
Thang Bui and Scott D. Stoller. 2020. A Decision Tree Learning Approach for Mining Relationship-Based Access Control Policies . In *25th ACM Symposium on Access Control Models and Technologies (SACMAT '20), June 10–12, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3381991.3395619

1 INTRODUCTION

In *relationship-based access control* (ReBAC), access control policies are expressed in terms of chains of relationships between entities. This increases expressiveness and often allows more natural policies. High-level access control policy models such as attribute-based access control (ABAC) and ReBAC are becoming increasingly widely adopted, as security policies become more dynamic and more complex. ABAC is already supported by many enterprise software products, using a standardized ABAC language such as XACML or a vendor-specific ABAC language. Forms of ReBAC are supported in popular online social network systems and are being studied and adapted for use in more general software systems as well.

High-level policy models such as ReBAC allow for concise and flexible policies and promise long-term cost savings through reduced management effort. The up-front cost of developing a ReBAC policy to replace an existing lower-level policy, such as access control lists or an RBAC policy, can be a significant barrier to adoption of ReBAC. *Policy mining* algorithms have the potential to greatly reduce this cost, by automatically producing a high-level policy from existing lower-level data; vetting and tweaking it is significantly less work than creating a high-level policy from scratch. There is a substantial amount of research on role mining, surveyed in [14, 25], and a small but growing literature on ABAC policy mining [12, 13, 20, 22, 24, 26, 31, 32] (surveyed in [14]) and ReBAC policy mining [7–10, 21].

The ReBAC policy mining problem as defined by Bui et al. [8, 10] is: Given information about the attributes of all entities in the system, and the set of currently granted permissions; Find a ReBAC policy that grants the same permissions using concise, high-level rules. For realistic datasets, the search space of possible policies is enormous. In traditional ABAC languages, such as XACML, each expression involves at most one attribute dereference. In ReBAC, an expression may contain a *path expression* representing a chain of attribute dereferences, and the search space grows exponentially in the path length.

This paper proposes new ReBAC policy mining algorithms, called DTRM (Decision Tree ReBAC Miner) and DTRM$^-$, based on decision tree learning. Decision trees are a natural basis for ReBAC policy mining because logic-based policy rules can be extracted from them much more easily than rules from neural networks, Bayes classifiers, etc. Also, a decision tree is a compact representation of ABAC (and ReBAC) policies that supports efficient policy evaluation [23, 29]. DTRM has two main phases: (1) learn an authorization policy in the form of a decision tree, using a modified version of the decision tree learning algorithm in Scikit [18], which is an optimized version of the well-known CART algorithm [5], and then extract a set of candidate authorization rules from the decision tree; (2) construct the mined policy by optionally eliminating negative conditions and constraints from the candidate rules (depending on whether the target policy language is ORAL2 or ORAL2$^-$, as discussed below) and then merging and simplifying the candidate rules. We selected Scikit's algorithm because it has been used successfully in a variety of application areas, and a patch for the above modification is available for it.

Our approach is general and could be used to mine policies in any ReBAC language. Our implementation produces policies in an extension of ORAL (Object-oriented Relationship-based Access-control Language), a ReBAC policy language developed by Bui et al. ORAL [7–10] or a similar language [21] is used in much of the published work on ReBAC policy mining.

SACMAT '20, June 10–12, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7568-9/20/06…$15.00
https://doi.org/10.1145/3381991.3395619

ORAL interprets ReBAC as object-oriented ABAC: relationships are expressed using attributes that refer to other objects, and path expressions built from chains of such attributes, as in object-oriented languages such as UML and Java. In ORAL, rules are built from *atomic conditions*, each of which is a condition on a single object—the subject (the entity making the access request) or resource (the entity to which access is requested)—and *atomic constraints*, each of which expresses a relationship between characteristics of the subject and the resource. An example of a condition is subject.employer = LargeBank. An example of a constraint is subject.department ∈ resource.project.departments.

The most recent version of ORAL, introduced in [7], supports two additional set comparison operators. We refer to that version as ORAL2, and we introduce ORAL2$^-$, an extension of ORAL2 with negative conditions and negative constraints. A *negative* condition or constraint is the negation of an atomic condition or constraint, e.g., subject.employer ≠ LargeBank or subject.department ∉ resource.project.departments. We give algorithms, called DTRM and DTRM$^-$, that mine policies in ORAL2 and ORAL2$^-$, respectively. The motivation for introducing ORAL2$^-$ is that negation is supported in some well-known ABAC languages, including XACML, and some ReBAC languages [3, 11, 19], and it sometimes allows more concise policies. We also support mining ORAL2 policies, i.e., policies without negation, for two reasons. First, some organizations may prefer policy languages without negation to reduce the chance of writing rules that grant excess permissions when new entities are added; for example, a rule with the condition subject.department ≠ MechEng may grant excess permissions to members of new departments, whereas a rule with the condition subject.department ∈ {ChemEng, ElecEng} will not. Second, mining of ORAL2 policies allows direct experimental comparison of our approach with FS-SEA* [7], a state-of-the-art ReBAC policy mining algorithm.

To demonstrate the benefits of our approach, we conducted an experimental comparison with two state-of-the-art ReBAC policy mining algorithms: FS-SEA* [7] and Iyer et al.'s algorithm [21]. The datasets used in our experiments include four sample policies, two large case studies based on policies of real organizations [15, 17], and several synthetic policies including the synthetic policies used in [7].

In summary, the main contribution of this paper is new ReBAC policy mining algorithms with two significant advantages over state-of-the-art ReBAC policy mining algorithms. (1) Our algorithms are significantly faster; specifically, they are more than 10× faster than FS-SEA* on several datasets, and are several times faster than Iyer et al.'s algorithm, while achieving comparable or better quality of the mined policies. The speedup generally increases with policy size hence is expected to be even larger for the larger datasets arising in practice. (2) DTRM$^-$ mines policies in a richer language than FS-SEA* and Iyer et al.'s algorithm; specifically, the language includes set comparison operators and negation.

2 RELATED WORK

We discuss related work on ReBAC and ABAC policy mining.

2.1 Related work on ReBAC policy mining

Bui et al. developed several ReBAC policy mining algorithms [7–10], the most recent and best of which is FS-SEA* [7]. As shown in Section 7, our algorithms are comparably effective at discovering the desired ReBAC rules, and are significantly faster; furthermore, DTRM$^-$ can mine policies in a richer language (with negation). Our algorithms are also simpler than FS-SEA*, which combines neural networks and a grammar-based genetic algorithm incorporating numerous heuristics and including two stages of evolutionary search. This is reflected in the sizes of the implementations. There is 3 KLOC of code in common (which we copied from FS-SEA*), plus an additional 13 KLOC for FS-SEA*, compared with an additional 6 KLOC for DTRM$^-$ (our more complicated algorithm).

Bui et al.'s policy mining algorithm in [9], which is a variant of the algorithm in [10], mines ReBAC policies from incomplete and noisy information about granted permissions [9]. Extending our algorithm to handle incomplete and noisy information is a direction for future work. Decision tree pruning methods, which are designed to avoid overfitting the input data, might be suitable for this.

Iyer et al. present algorithms, based on ideas from rule mining and frequent graph-based pattern mining, for mining ReBAC authorization policies and graph transition policies [21]. Their policy mining algorithm targets a policy language that is less expressive than ORAL2$^-$, because it lacks set comparison operators and negation; furthermore, unlike ORAL2, it does not directly support Boolean attributes. Set comparison operators are useful in practice: they are supported in XACML and used in all sample policies and case studies in [10]. Boolean attributes can be encoded in their framework, but this may require adding significant numbers of edges (connecting nodes or edges representing Boolean values to resources, since all paths referred to by a rule need to end at the resource being accessed), increasing the running time. They experimentally compare their policy mining algorithm with Bui et al.'s greedy algorithm (however, they misinterpreted some vaguely labeled output of the tool and incorrectly reported that the greedy algorithm in [10] achieved semantic similarity 0.9 for eWorkForce, while it actually achieves semantic similarity 1). In our experiments described in Section 7, our algorithms are faster and more effective.

2.2 Related work on ABAC Policy mining

Xu et al. proposed the first algorithm for ABAC policy mining [32] and a variant of it for mining ABAC policies from logs [31]. Medvet et al. developed the first evolutionary algorithm for ABAC policy mining [24]. Iyer et al. developed the first ABAC policy mining algorithm that can mine ABAC policies containing deny rules as well as permit rules [20]. Karimi et al. proposed an ABAC policy mining algorithm that uses unsupervised learning based on k-modes clustering [22]. Cotrini et al. proposed a new formulation of the problem of ABAC mining from logs and a practical algorithm, called Rhapsody, to solve it [13]. Rhapsody is based on APRIORI-SD, a machine-learning algorithm for subgroup discovery. Rhapsody can easily be extended to handle path expressions and therefore to support a form of ReBAC policy mining, but its running time is sensitive to the number of features and would be quite high for ReBAC mining except on small problem instances [7]. Cotrini et al. also developed a "universal" access control policy mining algorithm framework, which can be specialized to produce policy mining algorithms for a wide variety of policy languages [12]; the downside, based on their experiments,

is that the resulting algorithms achieve lower policy quality than customized algorithms for specific policy languages.

A top-down approach to ABAC policy mining has been pursued, aiming to extract ABAC policies from natural language documents using natural language processing and machine learning [1, 28].

3 POLICY LANGUAGE

Our policy language, which we call ORAL2^{-}, is Bui et al.'s ORAL2 (our name for it) [7], extended to allow negative conditions and constraints. We give a brief overview of the language, and refer the reader to [7] for details of ORAL2 and to [10] for details of the original version of ORAL, which ORAL2 extends. This overview is largely the same as in [7]. We include it to make this paper more self-contained, for the reader's convenience.

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

A *class model* is a set of class declarations. Each field has a *type*, which is a class name or "Boolean", and a *multiplicity*, which specifies how many values may be stored in the field and is "one" (also denoted "1"), "optional" (also denoted "?"), or "many" (also denoted "*", meaning any number). Boolean fields always have multiplicity 1. Every class implicitly contains a field "id" with type String and multiplicity 1. A *reference type* is any class name (used as a type). Like [7], we leave inheritance as a topic for future work.

An *object model* is a set of objects whose types are consistent with the class model and with unique values in the id fields. Let type(o) denote the type of object o. The value of a field with multiplicity "many" is a set. The value of a field with multiplicity "optional" may be a single value or the placeholder $\perp$ indicating absence of a value.

A *path* is a sequence of field names, written with "." as a separator. A *condition* is a set, interpreted as a conjunction, of atomic conditions or their negations. An *atomic condition* is a tuple $\langle p, op, val\rangle$, where p is a non-empty path, op is an operator, either "in" or "contains", and val is a constant value, either an atomic value (if op is "contains") or a set of atomic values (if op is "in"). For example, an object o satisfies $\langle$dept.id, in, {CompSci}$\rangle$ if the value obtained starting from o and following (dereferencing) the dept field and then the id field equals CompSci. In examples, conditions are usually written using mathematical notation as syntactic sugar, with "$\in$" for "in" and "$\ni$" for "contains". For example, $\langle$dept.id, in, {CompSci}$\rangle$ is more nicely written as dept $\in$ {CompSci}. Note that the path is simplified by omitting the "id" field since all non-Boolean paths end with "id" field. Also, "=" is used as syntactic sugar for "in" when the constant is a singleton set; thus, the previous example may be written as dept=CompSci.

A *constraint* is a set, interpreted as a conjunction, of atomic constraints or their negations. 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 five operators: equal, in, contains, supseteq, subseteq. Implicitly, the first path is relative to the requesting subject, and the second path is relative to the requested resource. The empty path represents the subject or resource itself. For example, a subject s and resource r satisfy $\langle$specialties, contains, topic$\rangle$ if the set s.specialties contains the value r.topic.

In examples, constraints are written using mathematical notation as syntactic sugar, with "=" for "equal","$\supseteq$" for "supseteq", and "$\subseteq$" for "subseteq".

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. A rule must satisfy several well-formedness requirements [10]. For a rule $\rho = \langle st, sc, rt, rc, c, A\rangle$, let sType($\rho$) = st, sCond(ρ) = sc, rType(ρ) = rt, rCond(ρ) = rc, con(ρ) = c, and acts(ρ) = A.

In example rules, paths in conditions and constraints that start from the subject and resource are prefixed with "subject" and "resource", respectively, to enhance readability. For example, the e-document case study [10, 16] involves a large bank whose policy contains the rule: A project member can read all sent documents regarding the project. Using syntactic sugar, this is written as
$\langle$ Employee, subject.employer = LargeBank, Document, true, subject.workOn.relatedDoc $\ni$ resource, {read}$\rangle$,
where Employee.workOn is the set of projects the employee is working on, and Project.relatedDoc is the set of sent documents related to the project.

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

An object o *satisfies* an atomic condition $c = \langle p, op, val\rangle$, denoted $o \models c$, if $(op = \text{in} \wedge \text{nav}(o, p) \in val) \vee (op = \text{contains} \wedge \text{nav}(o, p) \ni val)$. An object o *satisfies* a condition c, denoted $o \models c$, if it satisfies each atomic condition or negated atomic condition in c. Objects o_1 and o_2 *satisfy* an atomic constraint $c = \langle p_1, op, p_2\rangle$, denoted $\langle o_1, o_2\rangle \models c$, if $(op = \text{equal} \wedge \text{nav}(o_1, p_1) = \text{nav}(o_2, p_2)) \vee (op = \text{in} \wedge \text{nav}(o_1, p_1) \in \text{nav}(o_2, p_2)) \vee (op = \text{contains} \wedge \text{nav}(o_1, p_1) \ni \text{nav}(o_2, p_2)) \vee (op = \text{supseteq} \wedge \text{nav}(o_1, p_1) \supseteq \text{nav}(o_2, p_2))$. An object o *satisfies* a constraint c, denoted $o \models c$, if it satisfies each atomic constraint or negated atomic constraint in c.

An *SRA-tuple* is a tuple $\langle s, r, a\rangle$, where the "subject" s and "resource" r are objects, and a is an action, representing (depending on the context) authorization for s to perform a on r or a request to perform that access. An SRA-tuple $\langle s, r, a\rangle$ *satisfies* a rule $\rho = \langle st, sc, rt, rc, c, A\rangle$, denoted $\langle s, r, a\rangle \models \rho$, if $\text{type}(s) = st \wedge s \models sc \wedge \text{type}(r) = rt \wedge r \models rc \wedge \langle s, r\rangle \models c \wedge a \in A$. The *meaning* of a rule ρ, denoted $[[\rho]]$, is the set of SRA-tuples that satisfy it. The *meaning* of a ReBAC policy π, denoted $[[\pi]]$, is the union of the meanings of its rules.

4 PROBLEM DEFINITION

We adopt Bui et al.'s definition of the ReBAC policy mining problem. We present the core parts of the definition here, and refer the reader to [10] for more details and discussion.

An *access control list (ACL) policy* is a tuple $\langle CM, OM, Act, AU\rangle$, where CM is a class model, OM is an object model, Act is a set of actions, and $AU \subseteq OM \times OM \times Act$ is a set of SRA tuples representing authorizations. Conceptually, AU is the union of ACLs. An ReBAC policy π is *consistent* with an ACL policy $\langle CM, OM, Act, AU\rangle$ if they have the same class model, object model, actions, and $[[\pi]] = AU$.

Among the ReBAC policies consistent with a given ACL policy π_0, the most desirable ones are those that satisfy the following two criteria. (1) The "id" field should be used only when necessary, i.e., only when every ReBAC policy consistent with π_0 uses it, because uses of it make policies identity-based (like ACLs) and less general. (2) The policy should have the best quality as measured by a given policy quality metric Q_{pol}, expressed as a function from ReBAC policies to the natural numbers, with small numbers indicating high quality. This is natural for metrics based on policy size, which are the most common type.

The *ReBAC policy mining problem* is: given an ACL policy $\pi_0 = \langle CM, OM, Act, AU\rangle$ 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 [27] and later extended to ReBAC [10]. Minimizing policy size is consistent with usability studies showing that more concise access control policies are more manageable [2]. WSC is a weighted sum of the numbers of primitive elements of various kinds that appear in a rule or policy. WSC is defined bottom-up. 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 negated atomic condition or constraint c is $1 + \text{WSC}(c)$. The WSC of a rule ρ, denoted $\text{WSC}(\rho)$, is the sum of the WSCs of the atomic conditions and atomic constraints in it, plus the cardinality of the action set (more generally, it is a weighted sum of those numbers, but we take all of the weights to be 1). The WSC of a ReBAC policy π, denoted $\text{WSC}(\pi)$, is the sum of the WSC of its rules.

5 ALGORITHM

5.1 Phase 1: Learn Decision Tree and Extract Rules

A *feature* is an atomic condition (on the subject or resource) or atomic constraint satisfying user-specified limits on lengths of paths in conditions and constraints. We define a mapping from feature vectors to Boolean labels: given an SRA tuple $\langle s, r, a\rangle$, we create a feature vector (i.e., a vector of the Boolean values of features evaluated for subject s and resource r) and map it to true if the SRA tuple is permitted (i.e., is in AU) and to false otherwise. We represent Booleans as integers: 0 for false, and 1 for true. We train a decision tree to learn this classification (labeling) of feature vectors.

We decompose the problem based on the subject type, resource type, and action. Specifically, we learn a separate decision tree $DT_{C_s,C_r,a}$ to classify SRA tuples with subject type C_s, resource type C_r, and action a. We do this for each $\langle C_s, C_r, a\rangle$ such that AU contains some SRA tuple with a subject of type C_s, a resource of type C_r, and action a. The inputs to $DT_{C_s,C_r,a}$ are limited to the features appropriate for subject type C_s and resource type C_r, e.g., the path in the subject condition starts with a field in class C_s. The set of labeled feature vectors used to train $DT_{C_s,C_r,a}$ contains an element generated from each possible combination of a subject of type C_s (in the given object model) and resource of type C_r.

This decomposition by type is justified by the fact that all SRA tuples authorized by the same rule contain subjects with the same subject type and resources with the same resource type. A rule can authorize SRA tuples with different actions since the last component of a rule is a set of actions. The first phase of our algorithm learns rules containing a single action; the second phase attempts to merge similar rules with different actions into a single rule authorizing multiple actions.

As an optimization, we discard a feature if it has the same truth value in all of the labeled feature vectors used to train a DT; for example, if all instances of some type C in the given object model have the same value for a field f, then atomic conditions on field f are discarded.

We also detect sets of *equivalent features*, which are features that have the same truth value in all feature vectors labeled true used to train a particular DT. For each set of equivalent features, we keep the features with the lowest WSC and discard the rest. This is justified by that fact that the discarded features cannot appear in a policy with minimum WSC and consistent with AU.

Each internal node of a decision tree is labeled with a feature. Each outgoing edge of an internal node corresponds to a possible value of the feature (true or false). Each leaf node is labeled with an classification label (permit or deny). A feature vector is classified by testing the feature in the root node, following the edge corresponding to the value of the feature to reach a subtree, and then repeating this procedure until a leaf node is reached.

Figure 1 shows an example of a decision tree that represents a rule in an electronic medical record policy. The subject type, resource type and action are "Physician", "MedicalRecord" and "read", respectively. Internal nodes and leaf nodes are represented in the figure by unfilled and filled boxes, respectively. The rule specifies that only non-trainee physicians can read medical records which are associated to them. Formally, the rule is written as ⟨ Physician, subject.isTrainee = False, MedicalRecord, true, subject ∈ resource.physician, {read}⟩.

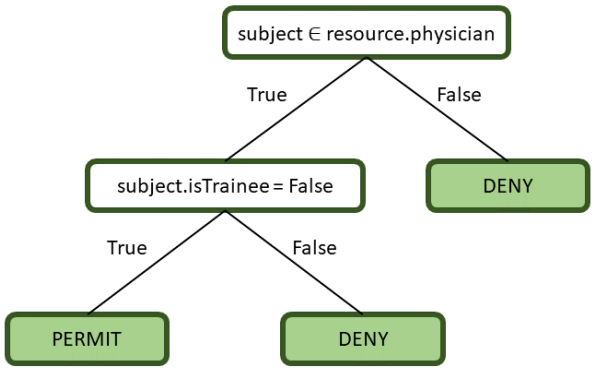

Figure 1: A sample decision tree for part of the healthcare sample policy.

5.1.1 Build Decision Trees. CART (and other well-known decision tree building algorithms including ID3 and C4.5) builds a decision tree by recursively partitioning feature vectors in the dataset, starting from a root node associated with the entire dataset. It chooses (as described below) a feature to test at the root node, creates a child node for each possible outcome of the test, partitions the set of feature vectors associated with the root node among the children, based on the outcome of the test, and recursively applies this procedure to each child. The recursion stops when all of the feature vectors associated with a node have the same classification label.

We use the decision tree learning algorithm in the Python library scikit-learn [18]. It is an optimized version of CART [5]. We disable its pruning methods. Pruning aims to reduce overfitting and make the decision trees generalize better. However, a pruned tree might misclassify some feature vectors in the training data. Pruning is therefore inappropriate for our purpose, which is to produce a policy completely consistent with the given ACL policy. The current implementation of the algorithm in scikit-learn treats categorical features as continuous features. For example, instead of treating a binary feature as a feature with possible values of 0 and 1, the test checks if the feature's value is less than or greater or equal than 0.5 for 0 and 1 respectively.

To choose which feature to test at each node n, the algorithm applies a scoring criteria to the remaining features (i.e., features that have not been used for splitting at an ancestor of n) and then chooses the top-ranked feature. The most popular scoring criteria are information gain and Gini index. For both of them, smaller values are better. We experimented with both on some sample policies, and the generated decision trees were identical. We adopted scikit-learn's default scoring metric, Gini index, for our experiments.

The Information Gain uses entropy to calculate the homogeneity of a set of feature vectors. Entropy is the measure of uncertainty of a random variable. The entropy is 0 if the sample contains only instances of the same class, and the entropy is 1 if the sample is equally divided. The information gain at node n for splitting with feature f is

$$InfoGain(n, f) = \sum_j \frac{|decis(S_n, f, j)|}{|S_n|} Entropy(decis(S_n, f, j))$$

$$Entropy(S) = -\sum_i \frac{|label(S, i)|}{|S|} \log_2 \frac{|label(S, i)|}{|S|}$$

where S_n is the set of feature vectors associated with the current node n, j ranges over the possible outcomes of testing feature f, $decis(S, f, j)$ is the subset of S containing feature vectors for which testing of feature f has outcome j, i ranges over the classification labels, and $label(S, i)$ is the subset of S containing feature vectors with label i.

The Gini Index uses impurity to measure how likely a randomly selected element would be misclassified. If all instances in the sample have the same class, the impurity will be 0. The Gini Index is calculated by subtracting the sum of squared probabilities of each class from 1. The Gini index for splitting at node n with feature f is

$$GiniIndex(n, f) = \sum_j \frac{|decis(S_n, f, j)|}{|S_n|} Impurity(decis(S_n, f, j))$$

$$Impurity(S) = 1 - \sum_i \left(\frac{|label(S, i)|}{|S|} \right)^2$$

When multiple features are tied for top-ranked according to the scoring criterion, scikit-learn chooses pseudorandomly among them. We adopt a modification to the algorithm that allows specification of a secondary metric as a tie-breaker, and we use the WSC of the feature as the secondary metric.

5.1.2 Extract rules. We convert the decision tree into an equivalent set of rules and include them in the candidate policy. For each distinct path through the tree from the root node to a leaf node labeled "PERMIT", we generate a rule containing the features associated with the internal nodes on that path; furthermore, if the path follows the False branch out of a node, then the feature associated with that node is added to the rule as a negative feature. For example, for the sample decision tree in Figure 1, only one rule is generated, which is the same as the input rule mentioned in Section 5.1. Rules extracted directly from the decision tress always have non-overlapping meanings. The next phase of our algorithm can produce rules with overlapping meanings.

5.2 Phase 2: Improve the Rules

Phase 2 has two main steps: eliminate negative features, and merge and simplify rules.

5.2.1 Eliminate Negative Features. This step is included only in DTRM, in order to mine rules without negation. This step is omitted from DTRM$^-$. This step eliminates each negative feature $\neg f$ in a rule ρ by applying the following substeps in order until one succeeds. A rule is *valid* if it covers only SRA tuples in AU.

(1) Remove $\neg f$ from ρ, if the resulting rule is valid.
(2) Replace $\neg f$ with a feature f', if the resulting rule is valid and the resulting policy (i.e., the policy with ρ replaced with the resulting rule) covers all SRA tuples in AU. In particular, try this for each feature f' not already used in ρ, in ascending order of WSC.
(3) If $\neg f$ is a negative atomic condition, and path p has multiplicity "one", then replace all of the negative atomic conditions with path p with a positive atomic condition using the same path p and the same operator, and with a set of constants which is the complement of the set of constants that appear in those negative atomic conditions. The complement is with respect to the set of all possible constants for that path. Note that this step always succeeds when it is applicable, i.e., the resulting rule is always valid, and the resulting policy always covers all SRA tuples in AU. Generalizing this step to apply when p has multiplicity "many" would require either replacing ρ with multiple rules, or extending the policy language to allow atomic conditions containing operators (such as $\supseteq$) for which both arguments have multiplicity "many".
(4) If $\neg f$ is a subject atomic condition, remove all subject atomic conditions (positive or negative) in ρ, and add the condition "subject.id $\in C$", where C is the set of ids of subjects that appear in SRA tuples covered by ρ. An analogous step applies if $\neg f$ is a resource atomic condition. Note that this step always succeeds when it is applicable.
(5) Replace $\neg f$ with a set of features, if the resulting rule is valid and the resulting policy covers all SRA tuples in AU. In particular, try this for all sets containing two more features not already used in ρ, in ascending order of WSC of the set (which

is the sum of the WSCs of the features in it). Note that this step can be reached only if f is a constraint. In the experiments described in Section 7, this step is never reached, i.e., one of the previous steps always succeeds.

5.2.2 Merge and Simplify Rules. This step attempts to merge and simplify rules using the same techniques as [10].

First, this step attempts to merge pairs of rules that have the same subject type, resource type, and constraint by taking the least upper bound of their subject conditions, the least upper bound of their resource conditions, and the union of their sets of actions. The *least upper bound* of conditions c_1 and c_2, denoted $c_1 \sqcup c_2$, is

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

When computing least upper bounds, DTRM$^-$ uses only positive atomic conditions; negative atomic conditions are dropped. 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.

Second, this step attempts to simplify the rules as follows.

(1) It eliminates atomic conditions from the subject and resource conditions when this preserves validity. Removing one atomic condition might prevent removal of another atomic condition, so it searches for a set of removable atomic conditions that maximizes the quality of the resulting rule.
(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.
(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 SRA tuples covered by a in ρ are covered by other rules in the policy. Note that the previous item is a special case of this one, listed separately to ensure that the 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.
DTRM$^-$ consider an additional case with the presence of negative condition/constraint. If the subject condition contains an atomic condition of the form $p = c$, and the constraint contains an atomic constraint of the form $p \neq p'$, then replace that atomic constraint with the atomic condition $p' \neq c$ in the resource condition; and similarly for the symmetric situation as mentioned in the first case.
(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 AU. A cycle is a path that navigates from some class C back to class C.
(7) If a subject/resource path of an atomic constraint evaluates to a same constant value c for all of the subjects/resources that are in SRA tuples covered by the rule, then replace the atomic constraint with corresponding resource/subject condition and constant c. In DTRM$^-$, if the atomic constraint is negative, it will be replaced with the corresponding negative atomic condition.

5.3 Asymptotic Running Time

This section analyzes the asymptotic running time of our algorithm. An extended version of this analysis appears in [6].

Phase 1. Let n_{feat} and n_{samp} be the number of features and feature vectors (samples), respectively. The cost of splitting samples at each node is $O(n_{\text{samp}} \cdot n_{\text{feat}})$.Let sz_{rule} be the "size" of the rules extracted from the tree, specifically, the sum of the numbers of features in each extracted rule; typically, the size of these intermediate rules is comparable to the size of the final mined rules. Note that the number of nodes in the tree is at most sz_{rule}. The cost of building the tree is $O(n_{\text{samp}} \cdot n_{\text{feat}} \cdot sz_{\text{rule}})$, and the cost of extracting the rules is $O(sz_{\text{rule}})$. This cost for each tree is summed over the number of trees, which is the number of $\langle C_s, C_r, a\rangle$ tuples, explained in Section 5.1.

Phase 2. The eliminating negative features step consists of several substeps, which are applied in order until one succeeds. Substep (1) takes $O(n_{\text{samp}})$ time, mainly for the rule validity check. Substep (2) takes $O(n_{\text{feat}} \cdot n_{\text{samp}})$ time to find the best valid replacement feature. Substep (3) takes $O(n_{\text{obj}})$ time, with n_{obj} is the maximum (over all types) number of objects of a single type in the object model. Substep (4) takes $O(n_{\text{samp}})$ time to compute the rule's coverage and extract the appropriate set of constants. We omit substep (5) from the complexity analysis here (but consider it in [6]), since this step is never reached in our experiments. Let n_{neg} be the number of negative features generated in the first phase; n_{neg} is typically small. If the first 4 substeps are all applied for every negative feature, the cost is $O(n_{\text{neg}} \cdot ((n_{\text{feat}} \cdot n_{\text{samp}}) + n_{\text{obj}}))$.

Let n_{rules} be the number of rules generated in Phase 1, and n_{cond} and n_{cons} be the maximum number of atomic conditions and atomic constraints, respectively, in each of these rules; n_{rules} is typically similar to the number of rules in the final mined policy. Let lm denote the maximum value of $|\,[[\rho]]\,|$ among all of the rules. The value of lm is at most $|AU|$ but typically much smaller. The cost of checking rule validity in these steps is $O(lm)$.

The merging step takes $O(n_{\text{rules}}^3 \cdot lm)$ time. The simplification step consists of several substeps. Substeps (1) and (2) take $O(n_{\text{rules}} \cdot 2^{n_{\text{cond}}} \cdot lm)$ and $O(n_{\text{rules}} \cdot 2^{n_{\text{cons}}} \cdot lm)$ time, respectively; the exponential factors here are small in practice, because rules typically have only a few conditions and constraints. Substeps (3) and (4) each take $O(n_{\text{rules}}^2 \cdot |Act| \cdot lm)$ time. Substep (5) takes $O(n_{\text{rules}} \cdot n_{\text{cond}} \cdot n_{\text{cons}})$ time. Substep (6) takes $O(n_{\text{rules}} \cdot (n_{\text{cond}} + n_{\text{cons}}) \cdot n_{\text{class}})$ time, where n_{class} is the number of classes in the class model. Substep (7) takes $O(n_{\text{rules}} \cdot n_{\text{cons}} \cdot lm)$ time.

6 EVALUATION METHODOLOGY

We adopt Bui et al.'s methodology for evaluating policy mining algorithms [7]. It is depicted in Figure 2. It takes a class model and a set

Policy_N	#obj	#field	#FV	#rule
EMR_15	353	877	4134	6
healthcare_5	736	1804	42121	8
project-mgmt_5	179	296	4080	10
university_5	738	926	83761	10
e-document_75	284	1269	31378	39
e-document_100	352	1653	52466	39
e-document_125	423	2065	82860	39
e-document_150	486	2406	108403	39
e-document_175	563	2830	152093	39
eWorkforce_10	412	1124	14040	19
eWorkforce_15	585	1647	31769	19
eWorkforce_20	691	1963	45625	19
eWorkforce_25	862	2484	74856	19
eWorkforce_30	1016	2928	104845	19
syn_20_x	678	7848	25600	x
syn_25_20	828	9773	40000	20
syn_30_20	978	11698	57600	20
syn_35_20	1128	13623	78400	20

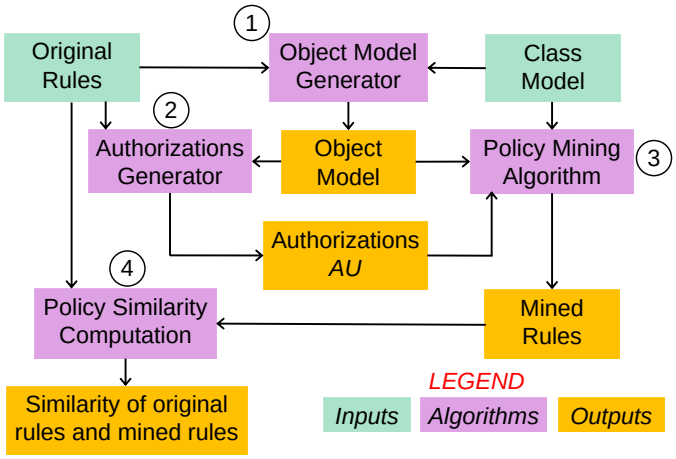

Figure 2: Left: Policy sizes. For the given value of the object model size parameter N, #obj is the average number of objects in the object model, and #field is the average number of fields in the object model, i.e., the sum over objects o of the number of fields in o. #FV is the number of feature vectors (i.e., labeled SRA tuples) that the algorithms use to train a classifier. Averages are over 5 pseudorandom object models for each policy. For the syn_20_M policies, the number of rules M is 10, 20, 30 or 40. Right: Evaluation methodology; reproduced from [7].

of ReBAC rules as inputs. The methodology is to generate an object model based on the class model (independent of the ReBAC rules), compute the authorizations AU from the object model and the rules, run the policy mining algorithm with the class model, object model, and AU as inputs, and finally compare the mined policy rules with the simplified original (input) policy rules, obtained by applying the simplifications in Section 5.2.2 to the given rules. 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. If the mined rules are similar to the simplified original rules, the policy mining algorithm succeeded in discovering the desired ReBAC rules that are implicit in AU.

6.1 Datasets

We use four sample policies developed by Bui et al. [10]. One is for electronic medical records (EMR), based on the EBAC policy in [4], translated to ReBAC; the other three are for healthcare, project management, and university records, based on ABAC policies in [32], generalized and made more realistic, taking advantage of ReBAC's expressiveness. These policies are non-trivial but relatively small.

We also use Bui et al.'s translation into ORAL2 [10] of two large case studies developed by Decat, Bogaerts, Lagaisse, and Joosen based on the access control requirements for Software-as-a-Service (SaaS) applications offered by real companies [15, 17]. One is for a SaaS multi-tenant e-document processing application; the other is for a SaaS workforce management application provided by a company that handles the workflow planning and supply management for product or service appointments (e.g., install or repair jobs).

Finally, we use the synthetic ORAL2 policies described in [7] and some extensions of them with additional rules.

All of the object models are generated by policy-specific pseudorandom algorithms designed to produce realistic object models, by creating objects and selecting their attribute values using appropriate probability distributions. These algorithms are 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. Bui et al.'s policy rules and object model generators for the sample policies and case studies, and their synthetic policy generator, are available online [30]. We slightly modified the object model generators for the project management, workforce management, and e-document policies, to make the generated object models slightly more realistic. More details about object model generation are in [7, 10].

These policies, or variants of them, have been used as benchmarks in several other papers on policy mining. In work on ReBAC mining, Iyer et al. [21] use variants of parts of three of the sample policies and the workforce management case study, and Bui et al. [9] use all of the sample policies and case studies. In work on ABAC mining, Medvet et al. [24], Iyer at al. [20], and Karimi et al. [22] use Xu et al.'s original ABAC versions of some of the sample policies.

The table in Figure 2 shows several metrics of the size of the rules, class model, and object model in each policy. #field is computed by summing, over the objects in the object model, the number of fields (including "id" field and Boolean fields) in each object.

The *Electronic Medical Record (EMR) sample policy*, based on the EBAC policy in [3], 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 numbers of physicians, consultations, EMRs, and hospitals are proportional to N.

The *healthcare sample policy*, based on the ABAC policy in [32], 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 [32], 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 [32], controls access by students, instructors, teaching assistants (TAs), department chairs, and staff in the registrar's office and admissions office to applications (for admission), gradebooks, transcripts, and course schedules. The numbers of departments, students, faculty, and applicants for admission are proportional to N.

The *e-document case study*, based on [15], is for a SaaS multitenant 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 [17], 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 only change we make is to omit from the workforce management case study the classes and 7 rules related to work orders, because they involve inheritance, which our algorithm does not yet support (it is future work). The numbers of helpdesk suppliers, workforce providers, subcontractors, helpdesk operators, contracts, work orders, etc., are proportional to N.

The *synthetic policies* developed by Bui et al. [7] are designed to have realistic structure, statistically similar in some ways to the sample policies and case studies described above. The class model is designed to allow generating atomic conditions and atomic constraints with many combinations of path length and operator. It supports the types of conditions and constraints that appear in the sample policies and case studies, plus constraints involving the additional constraint operators that are supported in ORAL2 but not in the original ORAL [10]. The object model generator's size parameter N specifies the desired number of instances of each subject class. The number of instances of each resource class is $5 \cdot N$. The numbers of instances of other classes is fixed at 3. This reflects a typical structure of realistic policies, in which the numbers of instances of some classes (e.g., doctors, patients, health records) scale linearly with the overall size of the organization, while the numbers of instances of other classes (e.g., departments, medical specialties) grow much more slowly (which we approximate as constant).

6.2 Policy Similarity Metrics

We evaluate the quality of the generated policy primarily by its *syntactic similarity* and *policy semantic similarity* to the simplified original policy. These metrics are first defined in [9, 32] and are normalized to range from 0 (completely different) to 1 (identical). We adapt the syntactic similarity metric to take negation into account. The metrics are based on Jaccard similarity of sets, defined by $J(S_1, S_2) = |S_1 \cap S_2| \,/\, |S_1 \cup S_2|$. For convenience, we extend J to apply to single values: $J(v_1, v_2)$ is 1 if $v_1 = v_2$ and 0 otherwise.

Syntactic similarity of policies measures the syntactic similarity of rules in the policies, based on the fractions of types, conditions, constraints, and actions that rules have in common. The *syntactic similarity of rules* is defined bottom-up as follows. For a possibly negated atomic condition ac, let $\text{sign}(ac)$, $\text{path}(ac)$, and $\text{val}(ac)$ denote its sign (positive or negative), its path, and its value (or set of values), respectively. Syntactic similarity of atomic conditions ac_1 and ac_2 is 0 if they contain different paths, otherwise it is the mean of $J(\text{sign}(ac_1), \text{sign}(ac_2))$, $J(\text{path}(ac_1), \text{path}(ac_2))$, and $J(\text{val}(ac_1), \text{val}(ac_2))$; we do not explicitly compare the operators, because atomic conditions with the same path must have the same operator, since the operator is uniquely determined by the multiplicity of the path. For a set S of atomic conditions, let $\text{paths}(S) = \{\text{path}(ac) \mid ac \in S\}$. For sets S_1 and S_2 of atomic conditions,

$$\text{syn}(S_1, S_2) = |\text{paths}(S_1) \cup \text{paths}(S_2)|^{-1} \sum_{ac_1 \in S_1, ac_2 \in S_2} \text{syn}_{\text{ac}}(ac_1, ac_2)$$

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 $\text{syn}(\rho_1, \rho_2) = \text{mean}(J(st_1, st_2), \text{syn}(sc_1, sc_2), J(rt_1, rt_2), \text{syn}(rc_1, rc_2), J(c_1, c_2), J(A_1, A_2))$.

The *syntactic similarity of policies* π_1 and π_2, $\text{syn}(\pi_1, \pi_2)$, is the average, over rules ρ in π_1, of the syntactic similarity between ρ and the most similar rule in π_2.

The *semantic similarity of polices* measures the fraction of authorizations that the policies have in common. Specifically, the *semantic similarity* of policies π_1 and π_2 is $J([\![\pi_1]\!], [\![\pi_2]\!])$.

7 EVALUATION RESULTS

This section presents the results of experiments comparing our algorithms with Bui et al.'s FS-SEA* algorithm [7] and Iyer et al.'s algorithm [21]. DTRM and DTRM$^-$ are implemented in Python, except that phase 2 step 2 (merge and simplify rules) uses the Java code from Bui et al.'s implementation of FS-SEA*, available at [30]. Experiments were run on Windows 10 on an Intel i7-6770HQ CPU. In summary, we find that: (1) compared with FS-SEA*, our algorithms are comparably effective at discovering the desired ReBAC rules, and are significantly faster, with the speedup exceeding 10× for several datasets and generally increasing with policy size, hence expected be even larger for the large datasets arising in practice; and (2) compared with Iyer et al.'s algorithm, our algorithms are several times faster, and produce policies that are the same size or smaller (fewer rules) and more similar to the original policies.

7.1 Comparison with FS-SEA*

We compared DTRM and DTRM$^-$ with FS-SEA* using the datasets described in Section 6.1. We use the same path length limits (*cf.* Section 5.1) as in [7, 10]. For the case studies, we generated policies with varying size (of the object model): $N = 10, 15, 20, 25, 30, 35$ for eWorkforce and $N = 75, 100, 125, 150, 175$ for e-document. For each size, we generated 5 pseudo-random object models. For synthetic policies, we generated two families of policies. Synthetic policies are designated by syn_N_M, where N is the object model size parameter, and M is the number of rules. The first family consists of 5 sets of $M = 20$ synthetic rules, and object models with sizes $N = 20, 25, 30$ (one of each size); we chose $M = 20$ because it is the average number of rules in the sample policies and case studies. The second family consists of sets of $M = 10, 30, 40$ synthetic rules (one of each size), and 5 object models with size $N = 20$. We ran DTRM, DTRM$^-$, and FS-SEA* on all of them, and average the results for the five policies with the same N and M. The standard deviations are reasonable, indicating that averaging over 5 object models for each data point is sufficient to obtain meaningful results.

7.1.1 Policy Similarity and WSC. All three algorithms always mine policies that grant exactly the same authorizations as the input ACL policies and thus achieve perfect *semantic similarity* for all datasets.

All algorithms achieve similar *syntactic similarity* when comparing mined rules with simplified original rules, as explained in Section 6. The minimum, median, and maximum (over all datasets) syntactic similarity achieved by each algorithm are: 0.91, 0.98, 1.0 for FS-SEA*; 0.90, 0.98, 1.0 for DRTM; and 0.90, 0.97, 1.0 for DTRM$^-$. The syntactic similarity achieved by DTRM and DTRM- are usually the same or better than that achieved by FS-SEA*, and in the worst cases in Table 1, are at most 2% and 4% lower, respectively. DTRM$^-$ achieved slightly lower syntactic similarity since the input policies do not use any negative atomic condition/constraint.

We report results for WSC in terms of the ratio of the WSC of the policy mined by DTRM or DTRM$^-$ to the WSC of the policy mined by FS-SEA*; thus a ratio below 1 means that DTRM or DTRM$^-$ produce a more concise policy than FS-SEA*. The minimum, median, and maximum (over all datasets) of this ratio are: 0.79, 1.0, 1.21 for DRTM, and 0.86, 1.01, 1.32 for DTRM$^-$. WSC of policies mined by DTRM$^-$ is not smaller than WSC of policies mined by DTRM, even though theoretically negation could allow more concise policies. This indicates that DTRM$^-$ sometimes produces policies that use negation even when it is not beneficial. This is not surprising, because when constructing the decision tree, the algorithm does not have a preference for using or avoiding negation.

Detailed results for policy similarity appear in Table 1. Detailed results for WSC appear in [6]. We conclude that all three algorithms produce policies with similar quality according to all three metrics.

7.1.2 Running Time. We report results for running time as the speedup relative to FS-SEA*, i.e., the ratio of the running time of each algorithm to the running time of FS-SEA*. Detailed results appear in Table 1. The results are summarized in the stacked bar chart in Figure 3. Each bar has three segments, representing three overlaid bars, each corresponding to an algorithm. The total height (as measured on the y-axis) of the top of each segment is the speedup of that algorithm. The first (black) segment is for FS-SEA*, so it always has height 1. The second (white) segment is for DTRM. The third (shaded) segment is for DTRM$^-$. For example, if DTRM achieved speedup 2.2 and DTRM$^-$ achieved speedup 4.4 for some policy, then the top of the black segment would be at height 1, the top of the white segment at height 2.2 (hence the white segment would be 1.2 units long), and the top of the shaded segment at height 4.4. This stacked bar chart format is suitable for reporting the speedups because, in all of our experiments, DTRM$^-$ is faster than DTRM, and DTRM is faster than FS-SEA*. The bars within each cluster other than the sample policy cluster are ordered left-to-right by increasing policy size, specifically by object model size for the e-doc., eWorkforce, and syn clusters, and by number of rules for the syn_20 cluster. Observe that speedup generally increases from left to right within those clusters, i.e., generally increases with policy size. A main reason that speedups for e-document and synthetic policies are larger than for eWorkforce and most sample policies is that the former policies have a larger number of rules per $\langle C_s, C_r, a\rangle$ tuple (explained in Section 5.1). DTRM (and DTRM$^-$) achieve larger speedups for such policies, because FS-SEA* repeats its expensive processing (feature selection and evolutionary search) for each generated rule, while DTRM performs its expensive processing (tree construction) once per $\langle C_s, C_r, a\rangle$ tuple and can quickly extract multiple rules from a tree.

Experiments with Sample Policies. DTRM and DTRM$^-$ spend most of the time in phase 1 to learn decision trees. The averaged running times spent on phase 2 are less than 1 second for EMR_15 and projectmangagment_5, and are less than 3 seconds for healthcare_5 and university_5. DTRM and DTRM$^-$ are faster than FS-SEA* on all of these policies. The average speedup is 2.17 for DTRM and 2.38 for DTRM$^-$. DTRM has similar running time as DTRM$^-$ on the sample policies, since only a few negative features are generated when learning decision trees for these policies, and they are not useful and hence are removed in the "*merge and simplify rules*" phase, so the negative feature elimination step in DTRM has no work to do.

Experiments with Case Study Policies. For eWorkforce, the average speedup is 1.70 for DTRM and 1.96 for DTRM$^-$. The negative feature elimination step in DTRM has little effect on the speedup, since the decision trees generated from the first phase do not contain many negative features. For e-document, the average speedup is 2.92 for DTRM and is 8.13 for DTRM$^-$, and the speedup for DTRM$^-$ increases with policy size. The difference in speedup is larger for

Policy	Syntactic Similarity						Running Time (sec)							
	FS-SEA*		DTRM		DTRM$^-$		FS-SEA*		DTRM		SpdUp	DTRM$^-$		SpdUp
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ		μ	σ	
EMR_15	0.99	0.01	0.99	0.01	0.99	0.01	96	7.37	56	0.30	1.70	53	5.55	1.82
healthcare_5	1.00	0.00	1.00	0.00	1.00	0.00	111	14.54	80	41.07	1.39	70	29.68	1.57
project-mgmt._5	1.00	0.00	1.00	0.00	1.00	0.00	6	0.45	2	1.62	3.88	2	0.39	4.07
university_5	1.00	0.00	1.00	0.00	1.00	0.00	271	21.98	159	64.02	1.70	131	44.32	2.07
e-doc._75	0.93	0.02	0.90	0.01	0.90	0.01	696	133.88	296	42.57	2.35	121	14.90	5.75
e-doc._100	0.94	0.01	0.91	0.03	0.90	0.02	1734	542.88	650	64.94	2.67	250	19.14	6.94
e-doc._125	0.93	0.01	0.92	0.01	0.93	0.02	3516	1415.93	1200	276.13	2.93	481	24.52	7.31
e-doc._150	0.91	0.01	0.94	0.01	0.94	0.00	6068	1202.25	2292	323.12	2.65	735	58.44	8.25
e-doc._175	0.92	0.01	0.93	0.01	0.93	0.01	15218	4535.45	3823	460.36	3.98	1227	68.07	12.41
eWorkforce_10	0.97	0.01	0.98	0.01	0.97	0.01	70	9.32	50	5.80	1.41	48	0.69	1.47
eWorkforce_15	0.95	0.02	0.98	0.02	0.97	0.02	287	37.68	182	13.62	1.58	176	2.06	1.63
eWorkforce_20	0.92	0.03	0.98	0.02	0.96	0.02	669	89.99	426	94.17	1.57	319	5.16	2.10
eWorkforce_25	0.95	0.04	0.97	0.02	0.95	0.02	1750	294.25	946	280.87	1.85	745	10.67	2.35
eWorkforce_30	0.97	0.02	0.97	0.02	0.95	0.02	3113	725.28	1492	312.45	2.09	1378	17.42	2.26
syn_20_10	0.99	0.00	0.99	0.01	0.99	0.01	938	348.24	166	52.16	5.66	142	12.42	6.61
syn_20_20	0.98	0.01	0.98	0.02	0.97	0.04	3129	887.18	309	88.93	10.11	256	24.97	12.22
syn_20_30	0.99	0.00	0.99	0.01	0.98	0.03	6303	1258.03	379	88.73	16.65	317	27.25	19.86
syn_20_40	0.99	0.01	0.98	0.02	0.97	0.03	11169	2812.94	435	69.75	25.67	370	14.52	30.21
syn_25_20	1.00	0.00	0.98	0.02	0.97	0.04	6494	2283.31	571	142.31	11.38	485	42.43	13.40
syn_30_20	1.00	0.00	0.99	0.01	0.99	0.01	11161	3396.60	898	82.72	12.43	861	75.63	12.96
syn_35_20	0.99	0.01	0.99	0.01	0.99	0.01	21758	7355.68	1419	138.17	15.33	1416	114.47	15.36

Table 1: Comparison of DTRM, DTRM$^-$, and FS-SEA*. μ and σ are the mean and standard deviation, respectively. SpdUp is the speed up, computed as the ratio of the running time of each of our algorithms to the running time of FS-SEA*.

e-document, because more negative features are generated in phase 1, so the negative feature elimination step in DTRM takes longer.

DTRM and DTRM$^-$ have lower average speedups on sample policies and eWorkforce, compared with the other policies (discussed next), because these policies are simpler, allowing FS-SEA* to have relatively good running time on them. In particular, FS-SEA* needs only one or a few iterations of feature selection and evolution to learn the rules for a given combination of subject type, resource type, and action, whereas for the more complicated policies, FS-SEA* typically needs more such iterations.

Experiments with Synthetic Policies. In experiments with the first family of synthetic policies, with $M = 20$ rules and varying object model size, the average speedup is 12.31 for DTRM and 13.49 for DTRM$^-$. For both DTRM and DTRM$^-$, the speedup generally increases with object model size; the 3% dip from syn_25_20 to syn_30_20 is not statistically significant (it's less than the σ).

In experiments with the second family of synthetic policies, with object model size $N = 20$ and varying number of rules, the average speedup is 17.48 for DTRM and 20.76 for DTRM$^-$. The speedups of both DTRM and DTRM$^-$ significantly increase with the number of rules: for DTRM, speedup increases from 5.66 with 10 rules to 25.67 with 40 rules; for DTRM$^-$, speedup increases from 6.61 with 10 rules to 30.21 with 40 rules.

7.2 Comparison with Iyer et al.'s Algorithm

We compare DTRM and DTRM$^-$ with Iyer et al.'s ReBAC mining algorithm [21] using modified versions of the eWorkforce datasets described in Section 7.1. We use Iyer et al.'s translation of a subset of the eWorkforce rules (used in experiments in [21]) as a starting point, and update it retain more of the original ORAL2 rules. We also modify the ORAL2 rules to exactly match (in meaning and structure, not syntax) the translated rules. We end up with 17 rules in each framework. Note that the original eWorkforce rules cannot be used directly: they need to be simplified, because Iyer et al.'s framework in [21] is less expressive than ORAL2. In particular, we eliminate Boolean attributes, and set comparison operators other than equality. We also simplify the object models in the eWorkforce_10 dataset by eliminating fields and classes not used in the modified rules. We implemented a translator that converts the simplified object models into Iyer et al.'s "system graph" representation. This enables us to run their system on significantly larger system graphs than used in any of the experiments (with any policy, not just eWorkforce) in [21]. We then compare the results of running our algorithms and their implementation of their algorithms.

When run on the modified eWorkforce_10 dataset, their system does not finish in a reasonable time (we used a timeout of 30+ minutes, since DTRM and DTRM$^-$ take less than a minute for this dataset) for some object models, and it returns errors, such as "MemoryError" and "IndexError: pop from empty list", for others. We reported these

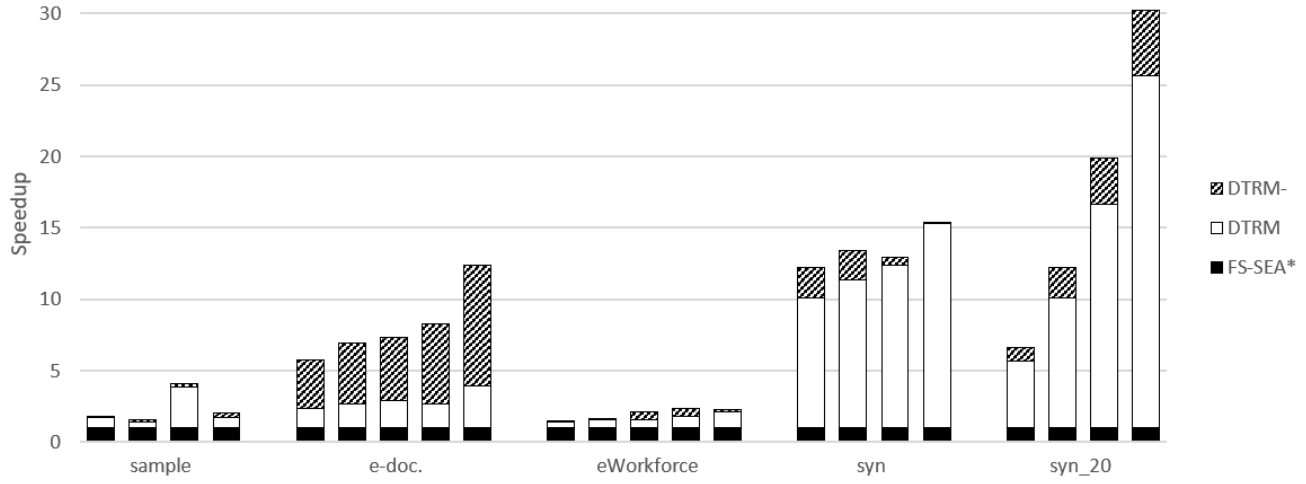

Figure 3: Speedups of DTRM and DTRM⁻ relative to FS-SEA*. There are 5 clusters, corresponding to 5 groups of policies. The "sample" cluster contains bars for the following policies (from left to right): EMR_15, healthcare_5, project-management_5 and university_5; "e-doc" cluster for e-document_75, e-document_100, e-document_125, e-document_150, e-document_175; "eWorkforce" cluster for eWorkforce_10, eWorkforce_15, eWorkforce_20, eWorkforce_25, eWorkforce_30; "syn" cluster for syn_20_20, syn_25_20, syn_30_20, syn_35_20; "syn_20" cluster for syn_20_10, syn_20_20, syn_20_30, syn_20_40.

Policy	Input Policies				Avg. # of Mined Rules			Avg. Running Time (sec)				
	#obj	#field	#FtVec	#rules	[21]	DTRM	DTRM⁻	[21]	DTRM	SpdUp	DTRM⁻	SpdUp
eWorkforce_10	354	530	8662	7	8	7	7	14	3	4.67	3	4.67
eWorkforce_15	505	751	20170	7	8	7	7	34	10	3.40	10	3.40
eWorkforce_20	601	897	29158	7	8	7	7	78	19	4.11	19	4.11
eWorkforce_25	755	1121	48253	7	8	7	7	101	42	2.40	41	2.46
eWorkforce_30	888	1304	67653	7	8.3	7	7	346	76	4.55	74	4.68

Table 2: Comparison of DTRM, DTRM⁻ and Iyer et al.'s algorithm on the simplified eWorkforce_10 dataset. #obj, #field, #FtVec and #rules have the same meanings as in Figure 2. SpdUp is the speedup of DTRM and DTRM⁻ relative to Iyer et al.'s algorithm.

issues to Iyer et al. Until they provide a fix, we circumvented these issues by removing the rules that trigger these issues, and removing parts of the object models unused by the remaining rules, until their system ran successfully for the remaining rules and at least one of the simplified object models for each object model size. In the end, we removed 10 rules that their system has trouble with, leaving 7 rules. The majority of the problematic rules are syntactically more complicated than the remaining ones. Specifically, 8 out of 10 of the problematic rules contain more than two atomic conditions/constraints (in ORAL2) or relationship patterns (in [21]'s policy language). In contrast, most (specifically, 5 out of 7) of the remaining rules contain only one atomic condition/constraint or relationship pattern (the other two remaining rules contain 3 atomic conditions/constraints). Results of these experiments are reported in Table 2. We set the path length limits for DTRM and DTRM⁻ to smaller values suitable for these simplified policies: MCSE = 5, MSPL = 2, MRPL = 1, SPED = 0, RPED = 0, and MTPL = 4. Even using these 7 remaining rules and significantly simplified object models, their system does not finish in a reasonable time (30 minutes) for some of the 5 object models for each policy size. Although we do not know for certain whether this is due to inefficiency of their algorithm or bugs in their implementation, we make the more generous assumption (i.e., assume the latter) and therefore omit those object models from the reported results. Consequently, the results in Table 2 are averages over 4 object models for eWorkforce_10, 1 for eWorkforce_15, 1 for eWorkforce_20, 2 for eWorkforce_25, and 3 for eWorkforce_30.

All three algorithms mine policies that grant the same authorizations as the input policies. For DTRM, the mined policies are identical to the input policies. For DTRM⁻, the mined policies are almost identical to the input policies: the only difference is replacement of the condition tenant.id = PP in one input rule with the negative condition tenant.id ≠ Telco, which is equivalent in context of these simplified object models. For Iyer et al.'s algorithm, the mined policy contain one more rule than the original policy (8 instead of 7) for all object models, except it contains two more rules for one object model of eWorkforce_30, because their algorithm fails to mine some of the

desired relationship patterns, generating instead multiple rules containing longer relationship patterns. We do not report WSC for these experiments, because the algorithms use different policy languages, and WSC is language-dependent.

DTRM and DTRM$^-$ are faster than Iyer et al.'s algorithm for all policies. Averaged over all policies, DTRM is 3.83 times faster, and DTRM$^-$ is 3.86 times faster. DTRM and DTRM$^-$ have very similar running times in these experiments, because very few negative features appear in the rules extracted from the decision trees.

8 FUTURE WORK

Directions for future work include: extending our algorithms to handle incompleteness and noise in the ACLs, perhaps using decision tree pruning methods, which are designed to avoid overfitting; extending our algorithms to identify errors in attribute values, and possibly suggest corrections; and developing incremental algorithms that efficiently handle updates to the object model or authorizations.

ACKNOWLEDGMENTS

This material is based on work supported in part by NSF grants CNS-1421893, CCF-1414078, and CCF-1954837 and ONR grant N00014-20-1-2751. We thank Hieu Le and R. Sekar for suggesting decision tree learning as an approach to policy mining. We thank Madison Ramos for valuable contributions to the initial stages of this work and the authors of [21] for sharing their code.

REFERENCES

[1] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. 2018. A Deep Learning Approach for Extracting Attributes of ABAC Policies. In *Proc. 23rd ACM on Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 137–148.

[2] Matthias Beckerle and Leonardo A. Martucci. 2013. Formal Definitions for Usable Access Control Rule Sets—From Goals to Metrics. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS)*. ACM, Article 2, 11 pages.

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

[4] Jasper Bogaerts, Maarten Decat, Bert Lagaisse, and Wouter Joosen. 2015. Entity-Based Access Control: supporting more expressive access control policies. In *Proc. 31st Annual Computer Security Applications Conference (ACSAC)*. ACM, 291–300.

[5] Leo Breiman, Jerome Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[6] Thang Bui and Scott D. Stoller. 2019. A Decision Tree Learning Approach for Mining Relationship-Based Access Control Policies. arXiv preprint arXiv:1909.12095 [cs.CR] (Sept. 2019).

[7] Thang Bui, Scott D. Stoller, and Hieu Le. 2019. Efficient and Extensible Policy Mining for Relationship-Based Access Control. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (SACMAT 2019)*. ACM, 161–172.

[8] Thang Bui, Scott D. Stoller, and Jiajie Li. 2017. Mining Relationship-Based Access Control Policies. In *Proc. 22nd ACM Symposium on Access Control Models and Technologies (SACMAT)*. 239–246.

[9] Thang Bui, Scott D. Stoller, and Jiajie Li. 2018. Mining Relationship-Based Access Control Policies from Incomplete and Noisy Data. In *Proceedings of the 11th International Symposium on Foundations & Practice of Security (FPS 2018) (Lecture Notes in Computer Science)*, Vol. 11358. Springer-Verlag.

[10] Thang Bui, Scott D. Stoller, and Jiajie Li. 2019. Greedy and Evolutionary Algorithms for Mining Relationship-Based Access Control Policies. *Computers & Security* 80 (jan 2019), 317–333. Preprint available at http://arxiv.org/abs/1708.04749. An earlier version appeared as a short paper in ACM SACMAT 2017.

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

[12] Carlos Cotrini, Luca Corinzia, Thilo Weghorn, and David Basin. 2019. The Next 700 Policy Miners: A Universal Method for Building Policy Miners. In *Proc. 2019 ACM Conference on Computer and Communications Security (CCS 2019)*. 95–112. https://doi.org/10.1145/3319535.3354196

[13] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC Rules from Sparse Logs. In *Proc. 3rd IEEE European Symposium on Security and Privacy (EuroS&P)*. 2141–2148.

[14] Saptarshi Das, Barsha Mitra, Vijayalakshmi Atluri, Jaideep Vaidya, and Shamik Sural. 2018. Policy Engineering in RBAC and ABAC. In *From Database to Cyber Security*. Lecture Notes in Computer Science, Vol. 11170. Springer Verlag, 24–54.

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

[16] Maarten Decat, Jasper Bogaerts, Bert Lagaisse, and Wouter Joosen. 2014. *The e-document case study: functional analysis and access control requirements*. CW Reports CW654. Department of Computer Science, KU Leuven.

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

[18] DT 2019. Decision Trees in scikit-learn v0.21.3. https://scikit-learn.org/stable/modules/tree.html.

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

[20] Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining Positive and Negative Attribute-Based Access Control Policy Rules. In *Proc. 23rd ACM on Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 161–172.

[21] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems. In *Proc. 24th ACM on Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 135–140.

[22] L. Karimi and J. Joshi. 2018. An Unsupervised Learning Based Approach for Mining Attribute Based Access Control Policies. In *2018 IEEE International Conference on Big Data (Big Data)*. 1427–1436. https://doi.org/10.1109/BigData.2018.8622037

[23] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. 2008. XEngine: a fast and scalable XACML policy evaluation engine. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*. ACM, 265–276.

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

[25] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2016. A Survey of Role Mining. *Comput. Surveys* 48, 4 (2016), 50:1–50:37. https://doi.org/10.1145/2871148

[26] Decebal C. Mocanu, Faith Turkmen, and Antonio Liotta. 2015. Towards ABAC policy mining from logs with deep learning. In *Proc. 18th International Information Society Multiconference (IS 2015), Intelligent Systems*. Institut Jozef Stefan, Ljubljana, Slovenia.

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

[28] Masoud Narouei, Hassan Takabi, and Rodney Nielsen. 2018. Automatic Extraction of Access Control Policies from Natural Language Documents. *IEEE Transactions on Dependable and Secure Computing* (2018).

[29] Ronit Nath, Saptarshi Das, Shamik Sural, Jaideep Vaidya, and Vijay Atluri. 2019. PolTree: A Data Structure for Making Efficient Access Decisions in ABAC. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (SACMAT 2019)*. ACM, 25–35.

[30] SSS 2019. Software from Scott Stoller's Research Group. https://www.cs.stonybrook.edu/~stoller/software/.

[31] Zhongyuan Xu and Scott D. Stoller. 2014. Mining Attribute-Based Access Control Policies from Logs. In *Proc. 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*. Springer, 276–291. Extended version available at http://arxiv.org/abs/1403.5715.

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