Mining Positive and Negative Attribute-Based Access Control Policy Rules

Padmavathi Iyer
University at Albany – SUNY
Albany, New York
riyer2@albany.edu

Amirreza Masoumzadeh
University at Albany – SUNY
Albany, New York
amasoumzadeh@albany.edu

ABSTRACT

Mining access control policies can reduce the burden of adopting more modern access control models by automating the process of generating policies based on existing authorization information in a system. Previous work in this area has focused on mining positive authorizations only. That includes the literature on mining role-based access control policies (which are naturally about positive authorization) and even more recent work on mining attribute-based access control (ABAC) policies. However, various theoretical access control models (including ABAC), specification standards (such as XACML), and implementations (such as operating systems and databases) support negative authorization as well as positive authorization.

In this paper, we propose a novel approach to mine ABAC policies that may contain both positive and negative authorization rules. We evaluate our approach using two different policies in terms of correctness, quality of rules (conciseness), and time. We show that while achieving the new goal of supporting negative authorizations, our proposed algorithm outperforms existing approach to ABAC mining in terms of time.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Authorization**;

KEYWORDS

attribute-based access control; policy mining; negative authorization; authorization conflicts

ACM Reference format:
Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining Positive and Negative Attribute-Based Access Control Policy Rules. In *Proceedings of The 23rd ACM Symposium on Access Control Models & Technologies (SACMAT), Indianapolis, IN, USA, June 13–15, 2018 (SACMAT '18),* 12 pages.
https://doi.org/10.1145/3205977.3205988

1 INTRODUCTION

Access control is one of the indispensable services of any information system responsible for protecting the underlying data from unauthorized access and inappropriate modifications [8]. *Attribute-based access control (ABAC),* as one of the more recent models for specifying access control policies, has been shown to overcome major limitations in previous models [10]. Unlike discretionary access control (DAC) or mandatory access control (MAC) models, ABAC is not dependent on user identities or rigid rules to determine authorizations as in DAC and MAC. Also, by allowing composition of flexible rules, it avoids problems such as role explosion [10] in role-based access control (RBAC). As the name suggests, ABAC models employ the attributes of users and resources to determine if an access request should be granted or denied, that is, attribute expressions are used to specify the sets of users and resources to which a policy is applicable [12, 22, 28]. For example, a policy such as *"A manager can read any document in his/her department"* may be directly translated to an ABAC policy: *"if userType=manager, resourceType=document, userDepartment=resourceDepartment, action=read then* `PERMIT`*"*. Thus, compared to traditional access control models, ABAC is very flexible in specifying access control policies, which makes ABAC a powerful access control model for promoting security.

We explore the problem of *mining ABAC policies* in this paper. Suppose an organization has already implemented some form of access control and wants to migrate to ABAC paradigm. Specifying ABAC policies manually from the existing access control information can be a time-consuming and error-prone job. To reduce the burden and expense of this task, the process of extracting policies from the given access control information can be partially or completely automated. This approach of automating the policy generation process is called *policy mining*. Mining RBAC policies, aka *role mining*, has been heavily studied in recent years [18]. Recently, Xu and Stoller have proposed an approach for mining ABAC policies [27, 26].

One of the limitations of policy mining approaches in the literature, addressing which is a central contribution of this paper, is lack of support for mining negative authorization rules. An ABAC policy can comprise of a set of positive and negative authorization rules, which grant or deny applicable access requests, respectively. Simultaneous use of positive and negative authorization rules are useful in situations when exceptions to more general rules need to be expressed, or when authorization rules with clonflicting outcomes originated from different viewpoints may have overlap. Beside ABAC, various other access control models in the literature allow expressing both positive and negative authorizations [13, 3, 9]. XACML policy language [6], a widely used standard for specifying and enforcing access control policies, also supports them. Negative authorizations have been also supported in products such as operating systems [21], web servers [1], and database systems [2].

SACMAT '18, June 13–15, 2018, Indianapolis, IN, USA

ACM ISBN 978-1-4503-5666-4/18/06…$15.00
https://doi.org/10.1145/3205977.3205988

Therefore, it is essential for an access control mining approach to be able to mine policies that may contain both positive and negative authorization rules. We note that although policy mining has been largely studied in the context of RBAC, due to lack of support for negative authorization in RBAC, naturally no previous work in that area addresses this problem [18]. The work by Xu and Stoller [27, 26], in the context of ABAC, also supports only mining policies with positive rules.

In this paper, we propose an algorithm for mining ABAC policies that can extract positive as well as negative authorization rules from a given access control information. Rather than designing an algorithm from scratch, we adopt an existing rule mining algorithm from the data mining literature, called *PRISM* [4], and extend that to capture positive and negative authorization rules simultaneously. Therefore, compared to previous work [27, 26], our algorithm provides a more systematic and less heuristic approach to mining access control rules that not only extracts negative authorizations but also performs better in terms of time. Our key contributions in this paper are as follows.

- We propose an algorithm for mining ABAC policies that, to the best of our knowledge, is a first of its kind approach to extract negative authorization rules in addition to positive authorization rules in the access control mining literature.
- We present a detailed approach to generate an authorization log that is needed as input to the mining algorithm, in case it is not readily available for a system.
- We implement a prototype and conduct experiments on the performance of our algorithm in terms of correctness and conciseness of the mined rules and the time taken to generate them. We demonstrate that, when the original (ground truth) policy includes negative authorization rules, our algorithm generates concise set of rules, which is not possible using previous work [27, 26]. Moreover, our algorithm outperforms previous work in terms of time in both cases of positive-only and positive-and-negative authorization policies.

The rest of the paper is organized as follows. Section 2 discusses our reference policy model that we use for specifying ABAC policies. In Section 3, we discuss about the design goals and challenges, and formally define the ABAC policy mining problem. Section 4 will describe the proposed ABAC policy mining algorithm in depth along with its time complexity. We discuss about an approach to generate complete authorization log in Section 5. In Section 6, we run experiments to test our proposed algorithm and analyze the results. Section 7 discusses related work in the field of policy mining and how our approach is novel compared to previous contributions. Finally, in Section 8, we provide additional discussions and conclusions.

2 ABAC POLICY MODEL

In this section, we present a specification and authorization semantics for ABAC policies that will be the basis for defining the policy mining problem and its proposed solution in this paper.

2.1 Policy Specification

An ABAC policy usually contains a disjunctive set of rules comprising attribute expressions on users and resources, actions, and applicable decisions. In the rest of this section, we generally use uppercase letters for denoting a set and lowercase letters for notating an element in a set.

Let U be the set of users in the system and R be the set of resources. A user is characterized by a set of attributes. Let $UATTR$ be the set of user attributes. To get the value of an attribute for a user, we use the notation $u.uattr$, where $uattr \in UATTR$ is the name of the user attribute. Like a user, a resource is characterized by a set of attributes, which will be denoted as $RATTR$. Given a resource attribute $rattr \in RATTR$, the value of that attribute for a resource is denoted by $r.rattr$. Let the *domain* of an attribute be the set of all possible values that the attribute can take. The domain of an attribute $attr \in UATTR \cup RATTR$ is denoted by $dom(attr)$. For simplicity, we consider only categorical attributes in this work. We assume that every user and resource in the system has a *unique identifier* attribute, defined by *uid* and *rid*, respectively. Authorizations in an ABAC system are determined for *actions* requested by users over resources. Let ACT be the set of all possible actions in the system.

The main components of an ABAC rule are attribute expressions that together (in a conjunctive format) determine the sets of users and resources to which a rule applies. An attribute expression can be either an *attribute-value* pair or an *attribute-attribute* pair. An attribute-value pair specifies the value corresponding to a user or resource attribute for the given rule to be applicable. An attribute-value pair for a user attribute *uattr* and a value *val* is expressed as $u.uattr = val$ and that for a resource attribute *rattr* and a value *val* is denoted as $r.rattr = val$. The followings are examples of attribute-value pairs:

- $u.department = CS$
- $r.type = transcript$

In the above examples, the first attribute-value pair indicates the set of users whose department is CS, while the second attribute pair indicates the set of resources whose type is transcript.

An attribute-attribute pair specifies a pair of user and resource attributes that need to match for the rule to be applicable. Formally, an attribute-attribute pair can be expressed as $u.uattr = r.rattr$, where $uattr \in UATTR$ and $rattr \in RATTR$. An example of attribute-attribute expression is as follows:

- $u.department = r.department$

In the above example, the attribute expression is satisfied by that set of users and resources where user department is the same as resource department.

Similar to attribute expressions, in particular attribute-value pairs, an ABAC rule includes an action expression that is denoted by $action = act$, where $act \in ACT$. Such an expression determines the access requests to which a rule will be applicable based on the requested action. Finally, each ABAC rule includes a rule effect, which is interpreted as granting applicable requests (`PERMIT`) or denying them (`DENY`). Given the abovementioned components, an *ABAC rule* is formally defined as a pair $\langle \phi, d \rangle$ where ϕ is a conjunctive set of attribute expressions and action expression and $d \in \{\texttt{PERMIT}, \texttt{DENY}\}$

is the rule effect. We use the following grammar for rule specification in this paper:

$$
\begin{aligned}
rule &::= \langle \phi,\ d \rangle \\
\phi &::= exp\ [;\ exp\ ] \\
exp &::= u.uattr = value \mid \\
&\quad\ r.rattr = value \mid \\
&\quad\ u.uattr = r.rattr \mid \\
&\quad\ action = value \\
d &::= \texttt{PERMIT} \mid \texttt{DENY}
\end{aligned}
$$

The followings are some examples of ABAC rules:

- $\langle u.position = faculty; u.chair = true; r.type = transcript; u.department = r.department; action = read\_transcript, \texttt{PERMIT}\rangle$
- $\langle u.position = manager; u.department = accounts; r.type = budget; action = approve, \texttt{DENY}\rangle$

The first example above is an example of a positive rule, according to which a user who is a faculty and chair of a department can perform read_transcript operation on all the transcripts in his/her department. On the other hand, the second example is an illustration of a negative rule, according to which if a manager from the accounts department tries to approve the budget of a project (project is a resource in this case), then he/she will be denied access.

As mentioned earlier, an ABAC policy is a disjunctive set of rules. We denote the complete set of rules in an ABAC policy by ρ. Along with the authorization rules, an ABAC policy also includes a *default decision* and a *conflict resolution strategy*. A *default decision* applies when none of the rules in the policy are applicable to an access request. A *conflict resolution strategy* applies when there is an overlap between positive and negative authorization rules. In other words, if both PERMIT and DENY rules are applicable to an access request, then the conflict resolution strategy of the policy decides the final decision for that access request [15, 16, 11].

In the context of this paper, in order to avoid overcomplicating our discussion about policy mining, we assume DENY as the default decision and *deny-overrides* as the conflict resolution strategy.

2.2 Authorization Process

An authorization request is a tuple $\langle u \in U, r \in R, a \in ACT\rangle$ indicating the requesting user, requested resource and action, respectively. Given an ABAC policy as described in Section 2.1, an ABAC authorization system evaluates each policy rule's expressions based on the request and determines if the rule is applicable to the request or not. There are three possible scenarios based on such a matching process:

- the access request matches with one or more rules in the policy, all of which contain the same access decision; or
- the access request matches with more than one rule in the policy but the access decisions of those rules are conflicting, i.e., include both PERMIT and DENY decisions; or
- the access request does not match with any rule in the policy.

The first scenario is quite straightforward. In this case, the authorization decision returned by the rule(s) in the policy. In the second scenario, the final authorization decision is resolved according to the conflict resolution strategy of the policy. For example, if the conflict resolution strategy of the policy is *deny-overrides*, then only one applicable DENY rule in the policy is sufficient to make the final authorization decision to be DENY. Finally, in the third scenario above, the default decision of the policy determines the authorization result. For example, if the default decision of the policy is DENY, then the system denies an access request if the request is not applicable with any of the rules in the policy.

3 PROBLEM STATEMENT

In this section, we present our design considerations and challenges for mining ABAC policies that include both positive and negative rules, and formulate the ABAC policy mining problem.

As input to the policy mining process, we consider a low-level log of authorization decisions in a system, which indicates authorization decision (PERMIT or DENY) for any given access by a user to a resource. Since our goal is to mine rules based on user and resource attributes, such a log needs to be accompanied (and augmented) by attributes of users and resources involved in the log entries. Such an access log may be accumulated by and retrieved from a working authorization system (e.g., as in collected in audit logs or for the sole purpose of mining). Also, in some cases, we may have an already existing access control policy specified using other models such as role-based access control (RBAC [7]) or simple access control lists (ACL [20]). In such cases, based on the existing policy, we may generate the desired log using an authorization engine or a log conversion process in case of simple models such as ACL.

A central design consideration and contribution of this work is coexistence of positive and negative rules in a mined policy. Ability to specify both positive and negative rules is desirable in cases such as handling simple exceptions (e.g., all but one department should be able to access a file) or implementing strict requirements for a group of users/resources (e.g., all employees on administrative leave should not be able to access any resource). This requirement brings on new challenges for mining ABAC policies compared to when mining positive rules only. In order to discuss better about the challenges, we illustrate two sample abstract policies as Venn diagrams in Figure 1. Here, the universe represents all possible access requests and corresponding decisions in the system. Each policy rule has been represented as a circle determining set of access instances to which it is applicable. As such, various overlapping situations can exist among policy rules. An overlap area indicates access instances with multiple applicable policies. As explained in Section 2.2, overlap can lead to a conflict situation if rules result in different decisions. For example, Figure 1(a) represents partial overlaps between two positive rules as well as overlap of the negative rules with the positive rules. Here, negative rules are proper subset of positive rules which may be used to specify exceptions to some permissions. For example, it can be the case that every student except students in the CS department can view the courselist:

- $\langle u.position = student; actions = view\_course\_list, \texttt{PERMIT}\rangle$
- $\langle u.position = student; u.department = CS; actions = view\_course\_list, \texttt{DENY}\rangle$

Figure 1(b) shows another example where a negative rule overlaps with multiple positive rules.

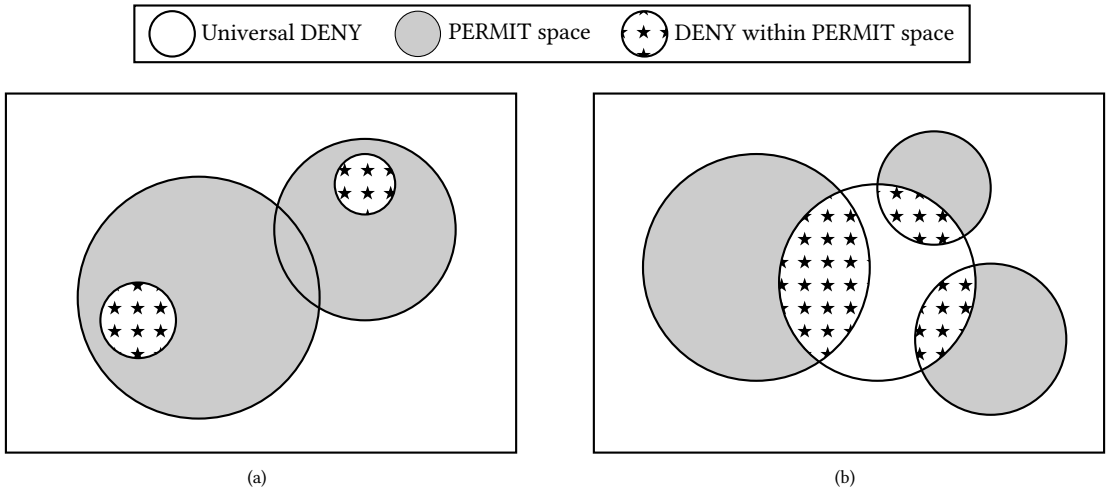

Figure 1: Policy spaces demonstrating `PERMIT` rules, conflicting `DENY` rules, and `DENY` space as default decision (a) Conflicting `DENY` rules are proper subset of `PERMIT` rules; and, (b) `DENY` rules conflict with more than one `PERMIT` rule.

Looking at the above examples from the viewpoint of a policy mining algorithm, which only sees a flat access log data, it is challenging to discover the rules when both positive and negative rules exist. The solution needs to discern `DENY` cases that are result of applying negative rules versus those that are result of applying the default rule. Note that we consider `DENY` as the default decision in this paper as explained in Section 2.1, . In Figure 1 the non-shaded area outside of the rules represents cases to which the default policy applies while the crossed areas represent cases when a negative rule results in `DENY`. Figure 1(b) highlights another desired characteristic for our solution. Rather than trying to generate three different specific negative rules, each corresponding to the crossed `DENY` pieces that are cut out of the positive rules, we need to be able to detect that they belong to one more general negative rule.

Finally, a policy mining solution should strive for deriving a policy that is as concise as possible as they are more manageable and easier to interpret. In terms of an ABAC policy, we would like to create less number of rules as well as creating rules with less number of expressions (more general rules). Previous work on mining ABAC policies [26] have adopted the notion of *Weighted Structural Complexity (WSC)*, previously defined in the context of RBAC policy mining [19], as a metric for this purpose. We adopt the same notion here. Informally, WSC of an ABAC policy is the sum of weights of all of its rules, where each rule's weight is calculated as the weighted sum of the number of expressions in that rule. Mathematically, WSC of an ABAC policy composed of the ruleset ρ is given as:

$$WSC(\rho) = \sum_{rule \in \rho} WSC(rule)$$

$$WSC(rule) = w_1|\alpha| \ + \ w_2|\beta| \ + \ w_3|\gamma|$$

where α, β and γ are, respectively, sets of attribute-value pairs, attribute-attribute pairs, and action expressions in rule's ϕ. Moreover, w_is are user-specified weights that adjust their contribution to rule's conciseness.

Based on the abovementioned considerations, we define the *ABAC policy mining problem* as follows. The ABAC policy mining problem accepts a complete authorization log (augmented with attributes) as input and extracts an ABAC policy (Section 2.1) that is concise and consistent with the authorization log. Based on the notations discussed in Section 2.1, the authorization log is a set of records, each indicating attribute values of a requesting user (*UATTR*), attribute values of requested resource (*RATTR*), an action (*ACT*), and corresponding access decision (`PERMIT` or `DENY`). In this work, we assume a complete log as input, meaning that every potential combination of attribute values are provided in the log (or otherwise assumed and set to be equal to the default `DENY` decision). As the correctness criterion, the mined policy must be consistent with the input authorization, i.e., the authorization of a log entry according to the mined policy and the semantics described in Section 2.2 must result in the same access decision as in the log entry. As the quality criterion, a solution to the mining problem must aim for policies that are as concise as possible. We quantify performance towards achieving this goal using the abovementioned WSC metric.

4 MINING ABAC POLICIES

In this section, we propose an approach for mining ABAC policies that include both positive and negative rules based on the policy model discussed in Section 2. Our proposed algorithm follows a systematic flow to mine optimal rules, as shown in Figure 2, avoiding heuristic and sub-optimal procedures as much as possible. The flow starts with mining positive rules, but also discovers conflicting negative rules simultaneously as a subprocess. In the following, we present our algorithm and analyze its time complexity.

4.1 Positive/Negative Rule Mining Algorithm

In order to mine attribute-based rules from authorization logs, we adopt concepts from a rule mining algorithm, called *PRISM* [4].

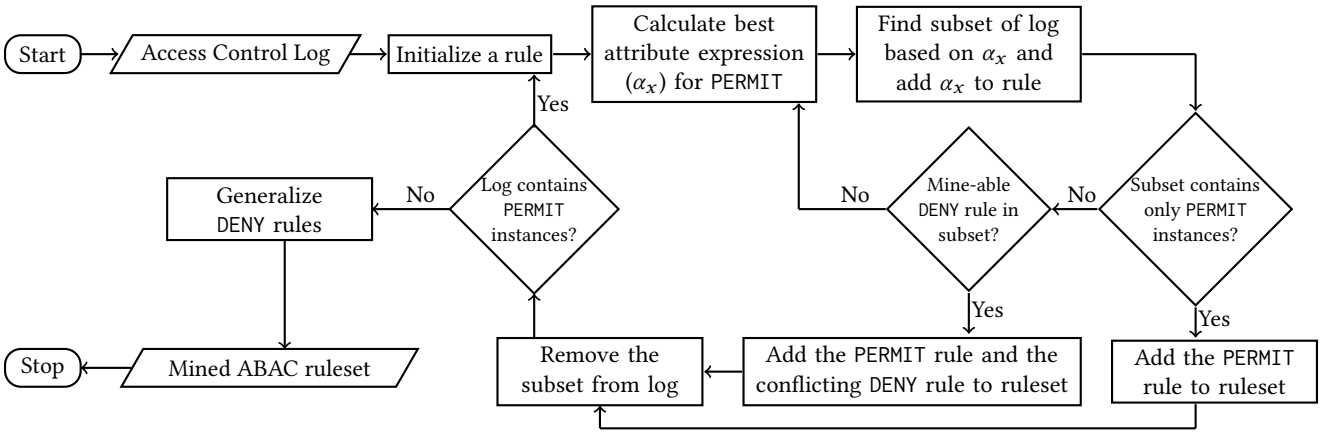

Figure 2: Flow chart of the proposed ABAC policy mining algorithm

The backbone of PRISM algorithm is induction strategy for finding the attribute-value pair, α_x, which yields highest conditional probability for a particular classification, δ_n, that is, for which $P(\delta_n \mid \alpha_x)$ is maximum. In context of this paper, conditional probability $P(\delta_n \mid \alpha_x)$ is the probability of occurrence of PERMIT or DENY decision, δ_n, for a given attribute expression, α_x.

At a high level our ABAC policy mining algorithm works in an iterative manner as shown in Algorithm 1. The input to the algorithm is an authorization log (augmented with user/resource attributes) as described in Section 3. The *getLastCol* function returns all the access decisions, in order, from the input dataset. The outer while loop runs until the log does not contain any PERMIT instances, to ensure that all positive rules have been mined. The inner while loop runs until a subset of the log contains all PERMIT instances or a conflicting DENY rule is encountered. Basically, the inner while loop is used to mine either a positive rule or a pair of positive and conflicting DENY rules. Within the inner loop, lines 7-17 returns the attribute expression, either attribute-value or attribute-attribute pair, that yields the highest conditional probability for PERMIT. If equal probabilities are encountered, then the one with larger coverage is returned. An attribute expression has larger coverage over another if the number of instances in the dataset that contains the former is greater than that for latter. The selected attribute expression is then added to the positive rule. *getInstances* function returns a subset of the log containing those instances that satisfy the selected attribute expression. Within this subset, existence of a conflicting DENY rule is checked using *findDenyRule* function (Algorithm 4). If it does, the conflicting DENY rule is added to ruleset and the inner loop breaks. Otherwise, inner loop repeats over the subset created in the previous iteration, until the subset comprises of only PERMIT instances. Lines 24-25 add PERMIT rule, which is created by taking the conjunction of all selected attribute expressions in the inner loop, to the ruleset, and remove all instances from the log that are covered by this rule. *generalizeDenyRules* function (Algorithm 5) generalizes all the negative rules in the ruleset by removing redundant attribute expressions from those rules. The output of the ABAC policy mining algorithm is a set of positive and conflicting negative rules.

Algorithm 1: mineRules

Input : *log* (complete authorization log)
Output: List of rules

1 $decision\_col \leftarrow getLastCol(log)$
2 **while** PERMIT $\in$ *decision_col* **do**
3 $X \leftarrow log$
4 $Y \leftarrow decision\_col$
5 $\phi \leftarrow \emptyset$
6 **while** DENY $\in Y$ **do**
7 $(attr, val, prob) \leftarrow$ findAttrValPair $(X,$ PERMIT$)$
8 $coverage1 = length(getInstances(X, attr, val))$
9 $(attr1, attr2, prob2) \leftarrow$ findAttrAttrPair $(X,$ PERMIT$)$
10 $coverage2 = length(getInstances(X, attr1, attr2))$
11 **if** $prob = prob2$ *and* $coverage1 > coverage2$ **then**
12 $(expr\_LHS, expr\_RHS) \leftarrow (attr, val)$
13 **else if** $prob2 < prob$ **then**
14 $(expr\_LHS, expr\_RHS) \leftarrow (attr, val)$
15 **else**
16 $(expr\_LHS, expr\_RHS) \leftarrow (attr1, attr2)$
17 ϕ.add$(expr_LHS = expr_RHS)$
18 $X \leftarrow getInstances(X, expr\_LHS, expr\_RHS)$
19 $Y \leftarrow getLastCol(X)$
20 $deny\_rule \leftarrow findDenyRule(X, \phi, Y)$
21 **if** $deny\_rule != null$ **then**
22 *ruleset*.add$(deny_rule)$
23 **break**
24 ruleset.add$(\langle \phi,$ PERMIT$\rangle)$
25 *log*.removeInstances(ϕ)
26 $decision\_col \leftarrow getLastCol(log)$
27 $ruleset \leftarrow generalizeDenyRules(ruleset, log)$

Algorithms 2 and 3 manifest two functions for returning attribute expressions, attribute-value pair and attribute-attribute pair, with highest conditional probability for PERMIT. The inputs to both these functions are the same. The loop in the *findAttrValPair* function

Algorithm 2: findAttrValPair

Input : X (log of access requests and decisions), *class* (PERMIT or DENY)
Output: Attribute-Value pair

1 $(maxProb, attr, val) \leftarrow (0, null, null)$
2 **for** $i \leftarrow 1$ **to** *(numAttributes(X)-1)* **do**
3 **foreach** $j \in getUniqueValues(X.getColumn(i))$ **do**
4 $prob \leftarrow P(class \mid i_j)$
5 **if** $maxProb < prob$ **then**
6 $(maxProb, attr, val) \leftarrow (prob, i, j)$
7 **else if** $maxProb = prob$ **then**
8 **if** $attr = null$ ***and*** $val = null$ **then**
9 $(attr, val) \leftarrow (i, j)$
10 **else if** $length(getInstances(X, attr, val)) < length(getInstances(X, i, j))$ **then**
11 $(maxProb, attr, val) \leftarrow (prob, i, j)$
12 **return** $(attr, val, maxProb)$

Algorithm 3: findAttrAttrPair

Input : X (log of access requests and decisions), *class* (PERMIT or DENY)
Output: Attribute-Attribute pair

1 $uAttr \leftarrow getUserAttributes(X)$
2 $rAttr \leftarrow getResourceAttributes(X)$
3 $(maxProb, attr1, attr2) \leftarrow (0, null, null)$
4 **for** $i \leftarrow 1$ **to** *(numAttributes(uAttr)-1)* **do**
5 **for** $j \leftarrow 1$ **to** *(numAttributes(rAttr)-1)* **do**
6 $prob \leftarrow P(class \mid [i, j])$
7 $actual\_i \leftarrow getActualIndex(i, X)$
8 $actual\_j \leftarrow getActualIndex(j, X)$
9 **if** $maxProb < prob$ **then**
10 $(maxProb, attr1, attr2) \leftarrow (prob, actual\_i, actual\_j)$
11 **else if** $maxProb = prob$ **then**
12 **if** $attr1 = null$ ***and*** $attr2 = null$ **then**
13 $(attr1, attr2) \leftarrow (actual\_i, actual\_j)$
14 **else if** $length(getInstances(X, attr1, attr2)) < length(getInstances(X, actual\_i, actual\_j))$ **then**
15 $(maxProb, attr1, attr2) \leftarrow (prob, actual\_i, actual\_j)$
16 **return** $(attr1, attr2, maxProb)$

enumerates all possible attribute-value pairs in the input dataset and calculates conditional probability in case of each attribute-value pair. The *getUniqueValues* function in line 3 returns the set of distinct values from the input set. The conditional probability in line 4 indicates the probability of occurrence of the given class (in this case, the class is PERMIT), given an attribute-value pair. If two attribute-value pairs have the same probability, then the one with higher coverage is selected (lines 7-11). The *findAttrAttrPair* function is similar to the *findAttrValPair* function, except that, instead of attribute-value pairs, the loop in *findAttrAttrPair* function enumerates all possible pairs of *uattr* and *rattr*, where $uattr \in UATTR$, $rattr \in RATTR$, and *uattr* and *rattr* have the same domain, that is, $dom(uattr) = dom(rattr)$. The conditional probability in line 6 (Algorithm 3) indicates the probability of occurrence of PERMIT, given an attribute-attribute pair.

Algorithm 4: findDenyRule

Input : X (log of access requests and decisions), ϕ (a permit rule), Y (list of decisions for access requests in X)
Output: A deny rule

1 $flag \leftarrow false$
2 $decision\_col \leftarrow getLastCol(X)$
3 **while** ***not***$(getUniqueValues(decision\_col) = \{\text{DENY}\})$ **do**
4 $(attr, val, prob) \leftarrow$ getAttrExp (X, DENY)
5 $coverage = length(getInstances(X, attr, val))$
6 $(expr\_LHS, expr\_RHS) \leftarrow (attr, val)$
7 **if** $getUniqueValues(X.getColumn(attr)) \equiv dom(attr)$ **then**
8 ϕ.add $(expr\_LHS = expr\_RHS)$
9 $flag \leftarrow true$
10 $X \leftarrow getInstances(X, rule\_LHS, rule\_RHS)$
11 $decision\_col \leftarrow getLastCol(X)$
12 **if** $flag = false$ **then**
13 **return** $(null)$
14 **else if** $coverage = length(getInstances(Y, \text{DENY}))$ **then**
15 **return** $(\langle \phi, \text{DENY} \rangle)$
16 **else**
17 **return** $(null)$

Algorithm 5: generalizeDenyRules

Input : *ruleset* (initial ruleset from mining algorithm), *log* (complete authorization log)
Output: Final ruleset with generalized deny rules

1 $covered\_instances \leftarrow \emptyset$
2 **foreach** $rule \in ruleset$ **do**
3 **if** $rule.d = \text{DENY}$ **then**
4 $coverage \leftarrow getRuleCoverage(rule)$
5 **if** $coverage \subseteq covered\_instances$ **then**
6 $ruleset.remove(rule)$
7 **else**
8 **foreach** $attrExp \in rule$ **do**
9 $gen\_cov \leftarrow getRuleCoverage(rule.remove(attrExp))$
10 **if** ***not***$(\text{PERMIT} \in gen\_cov)$ **then**
11 $rule \leftarrow rule.remove(attrExp)$
12 $ruleset.add(rule)$
13 $coverage \leftarrow getRuleCoverage(rule)$
14 $covered\_instances \leftarrow covered\_instances \cup coverage$
15 **return** *ruleset*

The *findDenyRule* function in Algorithm 4 mines a conflicting negative rule within a positive rule. The loop runs until all the instances in the input dataset contain only DENY. Within this loop,

the attribute expression yielding the highest conditional probability for DENY is selected. Lines 7-9 ensure that the selected attribute expression is added to the input rule only if the set of distinct values contained in attribute *attr* equals the domain of *attr*. The *flag* variable indicates whether any attribute expression was added to the input rule. A subset of the input dataset is created comprising of all instances containing the selected attribute expression. The loop is then repeated on this subset, until it contains only instances of DENY. At this point, a negative rule is created by taking the conjunction of all selected attribute expressions in the loop. The mined negative rule is indeed a conflicting negative rule if it covers all DENY instances in the input dataset (lines 12-17).

After generating the initial ruleset from the access control log, the policy mining algorithm generalizes the DENY rules in the ruleset as specified in Algorithm 5. For every DENY rule in the ruleset, we initially check if it is a subset of a generalized DENY rule, and if it is, then it is removed from the ruleset. Otherwise, the DENY rule is generalized by removing its components (attribute expressions) one at a time. Each time a component is removed, we check if the new DENY rule covers any PERMIT instances. If it does not, then the redundant component is removed from the original DENY rule. Finally, the generalized DENY rule is added to the ruleset. The *getRuleCoverage* function returns the set of instances covered by a rule in the access log.

4.2 Time Complexity

The time complexity of ABAC policy mining algorithm (Algorithm 1) can be calculated as follows. Let n be the number of records or instances and d be the number of attributes in the access log. The outer loop runs as many times as the number of PERMIT instances in the log. So, the running time of the outer loop is $O(n)$.

The inner loop runs as many times as the total number of attributes involved, including all the attribute expressions, within a particular PERMIT rule. In the worst case, a PERMIT rule can be formed by all attributes for attribute-value pairs and all combinations of attributes for attribute-attribute pairs. Since a rule cannot contain duplicate attribute expressions, total number of attributes included in attribute-value pairs is d and that for attribute-attribute pairs is of the order d^2. So, the running time of inner while loop is $O(d^2)$.

Calculating the optimal attribute-value pair (line 7 in Algorithm 1; Algorithm 2) takes $O(nd)$ in the worst case when all the attributes in the log contains n distinct values. Further, calculating the optimal attribute-attribute pair (line 9 in Algorithm 1; Algorithm 3) takes $O(d^2)$ time. So, total time taken for calculating the optimal attribute expression is $O(nd)$.

Computing a conflicting DENY rule (line 20 in Algorithm 1; Algorithm 4) takes total $O(nd^3)$ time. This is because the while loop in Algorithm 4 takes $O(d^2)$ time in the worst case when a DENY rule contains all attributes for attribute-value pairs and all combinations of attributes for attribute-attribute pairs. Moreover, calculating the optimal attribute expression (line 4 in Algorithm 4) takes $O(nd)$ time.

Generalizing the DENY rules (line 27 in Algorithm 1; Algorithm 5) consumes a total of $O(nd)$ time. This is because the loop in Algorithm 5 runs for each attribute, within every DENY rule in the initial ruleset. Since rules represent instances of the access log, the number of DENY rules in the initial ruleset is of the order n.

The total running time of our ABAC policy mining algorithm is, therefore, $O(n^2d^5)$. The time complexity of our policy mining approach is much less than $O(n^3)$, which is the worst case running time of [26] (details in Section 7). Suppose, every attribute in the log contains exactly two values in its domain, then total number of instances in a complete log, n, is 2^d. Besides, in a realistic application, domain of attributes have more than two values. So, for large values of d, $d^5 << m^d (= n)$, where $m \in \{2, 3, 4, ...\}$ depending on the application.

5 GENERATING ACCESS LOGS

In this part, we discuss the algorithm used for generating the log, in detail. The proposed algorithm can be used as a framework for generating synthetic logs, which can be utilized for various analysis purposes. For example, we use synthetic logs, generated from ABAC policies, for evaluation of our policy mining approach, so that we can have the ground truth while comparing the mined ABAC policy with original ABAC policy.

Log generation is an important phase in our proposed approach, because the log outputted from this phase serves as the input for the ABAC policy mining algorithm. Our goal is to understand the behavior of an underlying access control model. So we consider all possible combinations of all possible users, resources and actions, in short all possible scenarios of access requests, while generating the log, to be able to interpret all possible operations of the underlying access control model.

The log generation algorithm works as follows. Using the set of user attributes and domain for each user attribute, all possible users in the system are created by enumerating all possible combinations of values for user attributes. Similarly, all possible resources in the system are created using the set of resource attributes and domain for each resource attribute. A unique identifier is allocated to each user and resource. Then, using the complete set of users, resources and actions in the system, all possible *(user, resource, action)* combinations are determined, to enumerate all possible access requests that can be created from the system. While generating the complete set of access requests, each request is evaluated against the given XACML policy to determine the access decision for that request, following which the access request and corresponding access decision are written to a file.

Each record in the log indicates if a certain user can perform certain action on a certain resource. In other words, each row in the log contains the tuple *(A_u, A_r, Action, Decision)*, where A_u and A_r are, respectively, the set of requesting user and requested resource attributes, *Action* is the requested action, and *Decision* $\in$ {PERMIT, DENY } is the access decision corresponding to that access request.

5.1 Determining domains for each attribute

A challenge that we encountered while generating the log was to determine the domain for each user and resource attribute, because based on this domain information, all possible combinations of values for user and resource attributes can be created. In the context of this paper, we assume four types of columns or attributes: columns appearing only in the set of user attributes or resource

attributes (but not in both) referred to as *usr-only* attribute and *res-only* attribute respectively, columns that appear in the intersection of user and resource attribute sets referred as *usr-res* attributes, a user column dependent on the resource identifier column called as *usr-foreign-key* column, and a resource column dependent on the user identifier column referred to as *res-foreign-key* attribute.

The basic algorithm for defining the domain for all attributes is as follows:

Step 1: Determine the domain for all usr-only and res-only attributes.
Step 2: For every column $c \in$ usr-res, repeat:
- First determine the domain of c.
- Then the domain of each *uattr* and *rattr*, where $uattr \cap rattr = c$, is the same as the domain of c, that is, $dom(uattr) = dom(rattr) = dom(c)$.

Step 3: If the set of user attributes contains a column $c \in$ usr-foreign-key, then the domain of c, $dom(c)$, is the set or subset of resource identifiers, as required by c.
Step 4: If the set of resource attributes contains a column $c \in$ res-foreign-key, then the domain of c, $dom(c)$, is the set or subset of user identifiers, as required by c.

6 EVALUATION

We have implemented the proposed algorithms in Section 4 and report our experimental evaluation in this section. As our evaluation approach, rather than starting from an access log, we conduct our experiments by generating an access log from an ABAC policy, and then mine policies based on the generated log. Such an approach ensures that we have access to *ground truth* policies (i.e., original policies) with which we can compare the results of our mining algorithm. We follow a systematic approach to generate a comprehensive as well as minimal log as proposed in Section 5.

We compare the performance of our algorithm with the previously proposed algorithm by Xu and Stoller [26], which we refer to as `XSAM` in the rest of this section. We should note that `XSAM` is only capable of mining positive attribute-based access control policies. Therefore, we conduct our experiments on both policies that contain only positive rules, and policies that include positive as well as conflicting negative rules.

6.1 Datasets

We perform our experiments on two policy datasets that we have have adapted from [26] to include negative authorizations. The university policy, `University`, authorizes accesses to applications, gradebooks, rosters and transcripts, requested by students, faculties, applicants and staff in registrar/admissions office. The project management policy, `Project`, controls accesses by accountant, auditor, planner and manager to tasks, schedules and budgets associated with projects.

In order to provide a fair assessment and comparison of our algorithm versus `XSAM`, we use two different versions of `University` and `Project` policies. Policies `UniversityP` and `ProjectP` contain only postive authorization rules, while policies `UniversityPN` and `ProjectPN` have both positive and negeative authorization rules. We have included the policies in Appendix A.

6.2 Implementation

Our log generation implementation works based on list of all possible user attributes and resource attributes along with the domain for each attribute, list of all possible actions, and an ABAC policy written in XACML 3.0 [6] (Section 5). Each policy in XACML comprises of a set of rules, where each rule consists of a sequence of attribute expressions to determine which access requests the rule applies to and a rule effect to determine the access decision in case the rule is satisfied by an access request. Consistent with our policy model, we use *deny-overrides* rule-combining algorithm for XACML policies. The log generation algorithm is implemented in Java (JDK 1.8). We use WSO2 Balana [25], an open-source XACML implementation, to determine access decisions corresponding to each access request for a given XACML policy. The policy mining algorithm is written in Python (Python 3.5). We performed each experiment 10 times and report the average time measurement in our experiments. The experiments were performed on a 64-bit Windows 10 machine having 12 GB RAM and Intel Core i7-6700HQ processor.

Table 1 summarizes the access logs generated for university and project management policies. We note that the same number of access requests were generated regardless of positive-only vs. positive/negative policy versions. $|attr_u|$ is the number of user attributes, including the unique identifier attribute. Similarly, $|attr_r|$ is the number of resource attributes, including the identifier attribute. $|U|$, $|R|$ and $|O|$ are, respectively, the total number of user, resources and actions in the system. |log| is the total number of records in the generated log, which is computed as $|U|$ x $|R|$ x $|O|$.

6.3 Experiments with Positive Authorizations

We first compare the performance of our policy mining algorithm with XSAM [26] on policies consisting of only positive authorizations. This can provide an insight on how performances comapre on solving a mining problem that both approaches should be able to solve by design. We use the complete log as input to our algorithm, and provide only the `PERMIT` instances as input to `XSAM` since it works based on access control lists (ACLs).

The first four rows in Table 2 show the results of our algorithm and `XSAM` on `UniversityP` and `ProjectP` policies. The table compares approaches on the basis of quality, with respect to preciseness and conciseness, of mined rules and total time taken for execution. $|\rho_{orig+}|$ and $|\rho_{orig-}|$ are the number of positive and negative rules in the original ABAC policies, whereas $|\rho_{mined+}|$ and $|\rho_{mined-}|$ are the number of positive and negative rules in the mined ABAC policies. Further, WSC_{orig} and WSC_{mined} are, respectively, the WSC measure for original and mined policies. When calculating WSC for the experiments, we consider all user-specified weights w_i to be equal to one. Finally, *Run time* is the total time, in seconds, taken for mining ABAC policies from the given access control information.

As demonstrated in Table 2, both approaches, `XSAM` [26] and our proposed work, perform exactly the same in terms of mining concise rules that are syntactically and semantically similar to the original policy. However, our approach outperforms `XSAM` in terms of running time.

Table 1: Details of the access logs created from original ABAC policies

Policy	$\|attr_u\|$	$\|U\|$	$\|attr_r\|$	$\|R\|$	$\|O\|$	$\|\log\|$
University	6	128	5	2048	9	2359296
Project	8	4608	6	72	7	2322432

Table 2: Comparison of our proposed algorithm with XSAM [26] for university and project management policies

Mining Alg.	Policy	$\|\rho_{orig+}\|$	$\|\rho_{orig-}\|$	$\|\rho_{mined+}\|$	$\|\rho_{mined-}\|$	WSC_{orig}	WSC_{mined}	Time (s)
XSAM	UniversityP	5	-	5	-	19	19	1540
Proposed work	UniversityP	5	-	5	-	19	19	936
XSAM	ProjectP	11	-	11	-	49	48	1328
Proposed work	ProjectP	11	-	11	-	49	48	896
XSAM	ProjectPN	11	4	20	-	67	4324	1370
Proposed work	ProjectPN	11	4	11	4	67	64	1032
XSAM*	UniversityPN	11	3	-*	-*	56	-*	7200+*
Proposed work	UniversityPN	11	3	11	3	56	53	1123

* XSAM [26] did not terminate nor produced any output for the UniversityPN policy even after running for more than two hours.

6.4 Experiments with Positive and Negative Authorizations

Our second set of experiments is on policies consisting of both positive and negative authorization rules. The last four rows in Table 2 show the performance of the two approaches on ProjectPN and UniversityPN policies. Our observations show that our approach precisely mines concise positive and negative rules for both policies, whereas XSAM [26] computes verbose rules that are more identity-based rather than attribute-based. For example, in case of ProjectPN, while our proposed algorithm mines total of 15 rules with WSC of 64, XSAM produces 20 rules with significantly large WSC (4332). Furthermore, in our experiments, XSAM was not able to terminate and produce an output for UniversityPN even after running for more than two hours. The result clearly demonstrate the need for mining negative authorization rules along with positive rules.

6.5 Discussion of Results

6.5.1 Overall Analysis. As shown in Table 2, our policy mining algorithm precisely mines all the positive and negative rules from the generated log. Although the input access control log contains records belonging to both types of DENY spaces, the DENY space as a result of negative authorization rules and the default DENY space, our policy mining algorithm successfully mines all and only the required DENY authorization rules.

Our manual observation of the mined rules showed that they never reference any identity-based attributes like unique user identifier attribute and unique resource identifier attribute. Further, the experiment results verified that the mined ABAC policy is equivalent to the original ABAC policy. Moreover, the WSC of mined policy is constantly less than or equal to the WSC of original policy, i.e., $(WSC)_{\text{mined}} \leq (WSC)_{\text{original}}$. This manifests that our algorithm mines policies that are at least as concise as the original policies, while maintaining the semantic meaning of the original policies.

6.5.2 Comparison with XSAM. Our policy mining algorithm and the XSAM approach perform similar when only positive authorization rules need to be mined. However, there is a significance difference when both positive as well as negative authorization rules are considered. More particularly, when experimenting on policies containing negative authorization, XSAM either does not terminate in reasonable time (after two hours for UniversityPN) or produces verbose positive rules containing identifier attributes (which should be avoided for ABAC policies). In addition, our policy mining approach always runs faster than that of XSAM.

Although it may be argued that XSAM considers negative instances implicitly (i.e., any access not permitted is denied), it fails badly when considering both positive and negative rules as demonstrated in our experiments. This is because the policies ProjectPN and UniversityPN are particularly hard to express using only positive rules, which emphasizes the need for explicitly mining negative authorization rules along with positive authorization rules.

7 RELATED WORK

One of the areas in policy mining that received great research interest was role mining. The problem of role mining is to determine an optimal set of roles R from the user-permission assignments (UPA) for obtaining RBAC configuration that is equivalent to the given user-permission assignments, that is, decomposing the given UPA into User-role Assignments (UA) and role-Permission Assignments (PA) [18]. Vaidya at al. defined the Basic-RMP problem for finding a minimal set of roles from the given UPA [23]. Edge-RMP, a variant of Basic-RMP, aims to minimize, along with number of roles, |UA|+|PA| [14, 24]. |S| denotes the cardinality of a set S. Furthermore, Colantonio et al. presented a cost-based metric for mining optimal set of roles [5]. Another metric , proposed by Molloy et al., called *Weighted Structural Complexity* (WSC) [19], is the weighted sum of the number of elements in R, UA, PA and other components of an RBAC system. The aim of WSC optimization problem is to find an RBAC configuration, consistent with the given UPA, such that

WSC of the mined RBAC policy is minimized. Consistent with [26], we adopt the notion of WSC for measuring the complexity of mined ABAC policies.

The limitation of RBAC policy mining is that, for obtaining RBAC configuration from the given User Permission Assignments, role mining problems consider only the positive authorizations, in terms of what permissions are assigned to users, based on their roles. Our ABAC mining approach on the other hand, considers both positive and negative authorizations while obtaining ABAC policy.

Xu and Stoller were the first to introduce the concept of ABAC policy mining [26]. The motivation behind the idea of ABAC mining is to ease the burden of migration to ABAC framework from an existing access control paradigm, by partially automating the process of migration. At a high level, their policy mining algorithm works as follows. Initially, they generate an Access Control List (ACL), which they refer as the User Permission Relation, from an ABAC policy and attribute data. Then their policy mining algorithm, while iterating over the tuples in the given User Permission Relation, select a user permission tuple that is used as the seed for creating a candidate rule. This candidate rule is then generalized by replacing conjuncts in attribute expressions with constraints. The goal of their generalization process is to increase the coverage of the rule in terms of the additional tuples that can be covered by the rule in the User Permission Relation. The set of candidate rules, which altogether cover the entire ACL, is then optimized by removing redundant rules and merging pairs of rules. A rule is redundant if it covers instances in the User Permission Relation already covered by some other rule. Two different rules, having the same constraints, are merged by taking the union of conjuncts, in those rules, for every attribute. However, their algorithm does not deal with negative authorizations. Moreover the ABAC policy mining algorithm presented in [26] is very heuristic and complicated to interpret. Importantly, their running time is cubic in the size of the ACL, whereas the time complexity of our ABAC mining approach is much less than cubic time as explained in detail in Section 4.2.

Recently, Medvet et al. [17] proposed an evolutionary, separate and conquer approach for mining ABAC policies, using the same policy language and case studies as in [26]. In their work, a new rule is generated and the set of access requests decreases to a smaller size during each iteration. Similar to Xu and Stoller [26] and unlike our proposed approach, their work is not capable of mining negative authorization rules. Moreover, there is not much difference in terms of performance compared to [26]. Therefore, we only compare our performance against [26].

Our policy mining algorithm is closely related to the PRISM rule mining algorithm [4] by Cendrowsk. PRISM is an established data mining algorithm for inducing rules corresponding to a given dataset. It serves as a solution for the traditional data mining classification problem. Given a training dataset, containing different classifications, PRISM outputs a set of modular rules, where each rule contains combination of attribute-value pairs for arriving at a particular classification. To yield a set of disjunctive rules, PRISM uses an induction strategy for finding the attribute-value that delivers the most information about a particular classification. In other words, when determining a rule for a particular classification δ_n, PRISM finds the attribute-value pair α_x that gives the highest conditional probability for the classification δ_n, that is, PRISM selects the α_x for which the probability of occurrence of the classification δ_n, given the attribute-value pair α_x, is maximum. However, the limitation of PRISM in the context of this paper is that, for a particular rule, PRISM tends to find only attribute-value pairs, but not attribute-attribute pairs. As a result, when PRISM is run on the access control log, which serves as a suitable training dataset comprising of two classifications `PERMIT` and `DENY`, PRISM creates rules containing the identifier attributes like the unique user identifier attribute and unique resource identifier attribute. As a result, the output is verbose since it contains large number of rules. Our policy mining algorithm, although based on PRISM, overcomes this drawback by also considering attribute-attribute pairs, along with attribute-value pairs, while constructing a rule for `PERMIT` or `DENY`.

8 DISCUSSIONS AND CONCLUSIONS

In this paper, we proposed an algorithm for mining ABAC policies capable of discovering both positive and negative authorization rules simultaneously. While previous approaches in access control policy mining literature had focused on positive-only authorization rules (including more recent work on ABAC mining [26]), our work significantly contributes to the area by discovering negative authorization rules as well. We evaluated our policy mining algorithm on logs generated from two synthetic but realistic policies. Our observations from experiments show that the mined rules never reference identity-based attributes like user identifier and resource identifier attributes. Also, the results demonstrate that the mined rules are equivalent to the original ABAC policy, and that the mined policies are concise compared to them. Furthermore, we demonstrated that our approach outperforms previous ABAC mining algorithm [26] through the experiments and theoretical analysis.

Our mining algorithm attempts to mine positive and negative rules simultaneously. An alternative strategy would be to mine all possible positive rules first and then combine rules in a way to resolve in more general set of positive and negative rules. However, such an alternative strategy will lead to many granular positive rules which then need to be considered for generalization. Combining granular rules is a complex problem itself to solve optimally. We note that the previous ABAC mining approach [26] followed such an strategy (for positive rules only). But it relied on heuristics for generalization (by considering only pairs of rules). A main objective of design of our mining approach was to avoid such heuristic, suboptimal strategies.

The proposed mining algorithm is feasible to be employed in practice based on our experimental results and theoretical analysis. Running time of the algorithm was in the order of a few minutes for the synthetic policies. Theoretically, the time complexity of our policy mining algorithm depends on size of complete log, which is exponential to number of attributes. While we acknowledge this limitation, we note that it is applicable to any log mining algorithm that aims to avoid false positives/negatives. Moreover, we note that policy mining is inherently an offline and less time-sensitive task.

As future work, we plan to extend our approach to incorporate other ABAC features such as support for numerical data and other relational operators such as subset in attribute expressions. We will also explore algorithmic improvements, and more extensive quantitative analysis based on policies of different sizes.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and helpful suggestions that guided us in improving the final manuscript.

REFERENCES

[1] Apache Tutorials - Apache HTTP Server. URL: https://httpd.apache.org/docs/2.0/misc/tutorials.html.

[2] E. Bertino, P. Samarati, and S. Jajodia. An extended authorization model for relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):85–101, Jan. 1997. ISSN: 1041-4347.

[3] E. Bertino, S. Jajodia, and P. Samarati. A Flexible Authorization Mechanism for Relational Data Management Systems. *ACM Trans. Inf. Syst.*, 17(2):101–140, Apr. 1999. ISSN: 1046-8188.

[4] J. Cendrowska. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, Oct. 1, 1987. ISSN: 0020-7373.

[5] A. Colantonio, R. Di Pietro, and A. Ocello. A Cost-driven Approach to Role Engineering. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 2129–2136, New York, NY, USA. ACM, 2008.

[6] eXtensible Access Control Markup Language (XACML) Version 3.0. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[8] E. Ferrari. Access Control in Data Management Systems. *Synthesis Lectures on Data Management*, 2(1):1–117, Jan. 1, 2010. ISSN: 2153-5418.

[9] N. Gal-Oz, E. Gudes, and E. Fernández. A Model of Methods Access Authorization in Object-oriented Databases. In *Proc VLDB*, pages 52–61, Jan. 1, 1993.

[10] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[11] F. Huonder. Conflict Detection and Resolution of XACML Policies. *Master's thesis, University of Applied Sciences Rapperswil*, 2010.

[12] X. Jin, R. Krishnan, and R. Sandhu. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *IFIP Annual Conference on Data and Applications Security and Privacy*, LNCS, pages 41–55. Springer, Berlin, Heidelberg, July 11, 2012.

[13] M. A. Al-Kahtani and R. Sandhu. Rule-based RBAC with negative authorization. In *20th Annual Computer Security Applications Conference*, pages 405–415, Dec. 2004.

[14] H. Lu, J. Vaidya, and V. Atluri. Optimal Boolean Matrix Decomposition: Application to Role Engineering. In *2008 IEEE 24th International Conference on Data Engineering*, pages 297–306, Apr. 2008.

[15] A. Lunardelli, I. Matteucci, P. Mori, and M. Petrocchi. A prototype for solving conflicts in XACML-based e-Health policies. In *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*, pages 449–452, June 2013.

[16] M. St-Martin and A. P. Felty. A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 166–175, New York, NY, USA. ACM, 2016.

[17] E. Medvet, A. Bartoli, B. Carminati, and E. Ferrari. Evolutionary inference of attribute-based access control policies. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 351–365. Springer, 2015.

[18] B. Mitra, S. Sural, J. Vaidya, and V. Atluri. A Survey of Role Mining. *ACM Comput. Surv.*, 48(4):50:1–50:37, Feb. 2016. ISSN: 0360-0300.

[19] I. Molloy, N. Li, Y. A. Qi, J. Lobo, and L. Dickens. Mining Roles with Noisy Data. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, pages 45–54, New York, NY, USA. ACM, 2010.

[20] R. S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[21] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Trans. Comput. Syst.*, 7(3):247–280, Aug. 1989. ISSN: 0734-2071.

[22] D. Servos and S. L. Osborn. HGABAC: Towards a Formal Model of Hierarchical Attribute-Based Access Control. In *International Symposium on Foundations and Practice of Security*, LNCS, pages 187–204. Springer, Cham, Nov. 3, 2014.

[23] J. Vaidya, V. Atluri, and Q. Guo. The Role Mining Problem: A Formal Perspective. *ACM Trans. Inf. Syst. Secur.*, 13(3):27:1–27:31, July 2010. ISSN: 1094-9224.

[24] J. Vaidya, V. Atluri, Q. Guo, and H. Lu. Edge-RMP: Minimizing administrative assignments for role-based access control. *Journal of Computer Security*, 17(2):211–235, Jan. 1, 2009. ISSN: 0926-227X.

[25] WSO2 Balana. URL: https://github.com/wso2/balana.

[26] Z. Xu and S. D. Stoller. Mining Attribute-Based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing*, 12(5):533–545, Sept. 2015. ISSN: 1545-5971.

[27] Z. Xu and S. D. Stoller. Mining Attribute-Based Access Control Policies from Logs. In *IFIP Annual Conference on Data and Applications Security and Privacy*, LNCS, pages 276–291. Springer, Berlin, Heidelberg, July 14, 2014.

[28] X. Zhang, Y. Li, and D. Nalla. An Attribute-based Access Matrix Model. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 359–363, New York, NY, USA. ACM, 2005.

A POLICY DATASETS

In the following tables, we list the rules in the policies that we used in the experiments. Table 3 lists the rules in the `ProjectP` policy. In addition to those rules, the `ProjectPN` policy includes the `DENY` rules listed in Table 4. Tables 5 and 6 show the rules in the `UniversityP` and `UniversityPN` policies, respectively.

- $\langle u.adminRole = manager; r.type = budget; u.department = r.department; action = read$, PERMIT$\rangle$
- $\langle u.adminRole = manager; r.type = budget; u.department = r.department; action = approve$, PERMIT$\rangle$
- $\langle r.type = schedule; u.projectLed = r.project; action = read$, PERMIT$\rangle$
- $\langle r.type = budget; u.projectLed = r.project; action = read$, PERMIT$\rangle$
- $\langle r.type = schedule; u.projectLed = r.project; action = write$, PERMIT$\rangle$
- $\langle r.type = budget; u.projectLed = r.project; action = write$, PERMIT$\rangle$
- $\langle r.type = schedule; u.project = r.project; action = read$, PERMIT$\rangle$
- $\langle r.type = task; u.task = r.rid; action = setStatus$, PERMIT$\rangle$
- $\langle r.type = task; r.proprietary = false; u.project = r.project; u.expertise = r.expertise; action = read$, PERMIT$\rangle$
- $\langle r.type = task; r.proprietary = false; u.project = r.project; u.expertise = r.expertise; action = request$, PERMIT$\rangle$
- $\langle u.isEmployee = true; r.type = task; u.project = r.project; u.expertise = r.expertise; action = read$, PERMIT$\rangle$
- $\langle u.isEmployee = true; r.type = task; u.project = r.project; u.expertise = r.expertise; action = request$, PERMIT$\rangle$
- $\langle u.adminRole = auditor; r.type = budget; u.project = r.project; action = read$, PERMIT$\rangle$
- $\langle u.adminRole = accountant; r.type = budget; u.project = r.project; action = read$, PERMIT$\rangle$
- $\langle u.adminRole = accountant; r.type = budget; u.project = r.project; action = write$, PERMIT$\rangle$
- $\langle u.adminRole = accountant; r.type = task; u.project = r.project; action = setCost$, PERMIT$\rangle$
- $\langle u.adminRole = planner; r.type = schedule; u.project = r.project; action = write$, PERMIT$\rangle$
- $\langle u.adminRole = planner; r.type = task; u.project = r.project; action = setSchedule$, PERMIT$\rangle$

Table 3: `ProjectP` policy rules

- $\langle u.adminRole = manager; u.department = dept2; r.type = budget; action = read$, DENY$\rangle$
- $\langle u.adminRole = manager; r.type = budget; r.project = proj21; action = approve$, DENY$\rangle$
- $\langle u.adminRole = planner; u.department = dept3; u.expertise = testing; r.type = schedule; action = read$, DENY$\rangle$
- $\langle r.type = task; r.department = dept2$, DENY$\rangle$

Table 4: `DENY` rules in `ProjectPN` policy. `PERMIT` rules are the same as in `ProjectP` (Table 3)

- $\langle r.type = gradebook; u.courseTaken = r.course; action = readScore$, PERMIT$\rangle$
- $\langle r.type = gradebook; u.courseTaught = r.course; action = readScore$, PERMIT$\rangle$
- $\langle u.position = faculty; r.type = gradebook; u.courseTaught = r.course; action = assignGrade$, PERMIT$\rangle$
- $\langle u.position = student; r.type = transcript; u.uid = r.student; action = readTranscript$, PERMIT$\rangle$
- $\langle u.position = faculty; u.isChair = true; r.type = transcript; u.department = r.department; action = readTranscript$, PERMIT$\rangle$

Table 5: `UniversityP` policy rules

- $\langle r.type = gradebook; u.courseTaken = r.course; action = readMyScores$, PERMIT$\rangle$
- $\langle r.type = gradebook; u.courseTaught = r.course; action = addScore$, PERMIT$\rangle$
- $\langle r.type = gradebook; u.courseTaught = r.course; action = readScore$, PERMIT$\rangle$
- $\langle u.position = faculty; r.type = gradebook; u.courseTaught = r.course; action = changeScore$, PERMIT$\rangle$
- $\langle u.position = faculty; r.type = gradebook; u.courseTaught = r.course; action = assignGrade$, PERMIT$\rangle$
- $\langle u.isChair = true; r.type = gradebook; u.department = r.department; action = readScore$, PERMIT$\rangle$
- $\langle r.type = gradebook; u.courseTaken = r.course; action = addScore$, DENY$\rangle$
- $\langle r.type = gradebook; u.courseTaken = r.course; action = readScore$, DENY$\rangle$
- $\langle r.type = gradebook; u.courseTaken = r.course; action = changeScore$, DENY$\rangle$
- $\langle r.type = gradebook; u.courseTaken = r.course; action = assignGrade$, DENY$\rangle$
- $\langle u.department = registrar; r.type = roster; action = read$, PERMIT$\rangle$
- $\langle u.department = registrar; r.type = roster; action = write$, PERMIT$\rangle$
- $\langle u.position = faculty; r.type = roster; u.courseTaught = r.course; action = read$, PERMIT$\rangle$
- $\langle r.type = transcript; u.uid = r.student; action = read$, PERMIT$\rangle$
- $\langle u.position = student; u.department = dept1; r.type = transcript; action = read$, DENY$\rangle$
- $\langle u.isChair = true; r.type = transcript; u.department = r.department; action = read$, PERMIT$\rangle$
- $\langle u.department = registrar; r.type = transcript; action = read$, PERMIT$\rangle$
- $\langle r.type = application; u.uid = r.student; action = checkStatus$, PERMIT$\rangle$
- $\langle u.department = admissions; r.type = application; action = read$, PERMIT$\rangle$
- $\langle u.department = admissions; r.type = application; action = setStatus$, PERMIT$\rangle$
- $\langle u.department = admissions; r.type = application; r.department = dept2; action = read$, DENY$\rangle$
- $\langle u.department = admissions; r.type = application; r.department = dept2; action = setStatus$, DENY$\rangle$

Table 6: `UniversityPN` policy rules