Mining Least Privilege Attribute Based Access Control Policies

Matthew W Sanders
Colorado School of Mines
Golden, Colorado
mwsanders@mines.edu

Chuan Yue
Colorado School of Mines
Golden, Colorado
chuanyue@mines.edu

ABSTRACT

Creating effective access control policies is a significant challenge to many organizations. Over-privilege increases security risk from compromised credentials, insider threats, and accidental misuse. Under-privilege prevents users from performing their duties. Policies must balance between these competing goals of minimizing under-privilege vs. over-privilege. The Attribute Based Access Control (ABAC) model has been gaining popularity in recent years because of its advantages in granularity, flexibility, and usability. ABAC allows administrators to create policies based on attributes of users, operations, resources, and the environment. However, in practice, it is often very difficult to create effective ABAC policies in terms of minimizing under-privilege and over-privilege especially for large and complex systems because their ABAC privilege spaces are typically gigantic. In this paper, we take a rule mining approach to mine systems' audit logs for automatically generating ABAC policies which minimize both under-privilege and over-privilege. We propose a rule mining algorithm for creating ABAC policies with rules, a policy scoring algorithm for evaluating ABAC policies from the least privilege perspective, and performance optimization methods for dealing with the challenges of large ABAC privilege spaces. Using a large dataset of 4.7 million Amazon Web Service (AWS) audit log events, we demonstrate that our automated approach can effectively generate least privilege ABAC policies, and can generate policies with less over-privilege and under-privilege than a Role Based Access Control (RBAC) approach. Overall, we hope our work can help promote a wider and faster deployment of the ABAC model, and can help unleash the advantages of ABAC to better protect large and complex computing systems.

CCS CONCEPTS

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

KEYWORDS

ABAC, Principle of Least Privilege, Machine Learning, Rule Mining

ACM Reference Format:
Matthew W Sanders and Chuan Yue. 2019. Mining Least Privilege Attribute Based Access Control Policies. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359805

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359805

1 INTRODUCTION

Access control is a key component of all secure computing systems but creating effective policies is a significant challenge to many organizations. Access control policies specify which privileged entities can exercise certain operations upon certain objects under various conditions. Too much over-privilege increases the risk of damage to a system via compromised credentials, insider threats, and accidental misuse. Policies that are under-privileged prevent users from performing their duties. Both of these conflicting goals are expressed by the Principle of Least Privilege (PoLP) which requires every privileged entity of a system to operate using the minimal set of privileges necessary to complete its job [15]. The PoLP is a fundamental access control principle in information security [19], and is a requirement in security compliance standards such as the Payment Card Industry Data Security Standard (PCI-DSS), Health Insurance Portability and Accountability Act (HIPAA), and ISO 17799 Code of Practice for Information Security Management [18].

Many access control models have been introduced to address the challenges of administrating policies, with different approaches to balance between the goals of ease of use, granularity, flexibility, and scalability. Access control models are constantly evolving, but Attribute Based Access Control (ABAC) continues to gain in popularity as a solution to many use cases because of its flexibility, usability, and ability to support information sharing. ABAC allows security policies to be created based on the attributes of the user, operation, and the environment at the time of an access request.

The flexibility of ABAC policies is both a strength and a hindrance. With the ability to create policies based on many attributes, administrators face difficult questions such as what constitutes "good" ABAC policies, how to create them, and how to validate them? Additionally, the ABAC privilege space of a system can be extremely large, so how can administrators determine which attributes are most relevant in their systems?

We address these issues by taking a rule mining approach to automatically create ABAC policies from systems' audit logs. Rule mining methods are a natural fit for creating ABAC policies which contain rules regarding the actions that users can perform upon resources under certain conditions. By identifying usage patterns from audit logs to automatically generate and evaluate ABAC rules, our approach will help an organization continuously improve its deployed policy under the guidance of PoLP. Using out-of-sample validation to evaluate the generated policies on a dataset of 4.7M Amazon Web Service (AWS) audit log events [2], we show that our rule mining algorithm is effective at generating policies which minimize both under-privilege and over-privilege assignment errors.

We make the following contributions in this paper: 1) a definition of the ABAC Privilege Error Minimization Problem ($PEMP_{ABAC}$) for balancing between under- and over-privilege errors in security policies, 2) an algorithm for automatically generating least

privilege ABAC policies by mining audit logs, 3) an algorithm for scoring ABAC policies using out-of-sample validation, 4) feature selection, scalability, and performance optimization methods for processing large ABAC privilege spaces, 5) a quantitative analysis of the performance of our mining algorithm using a real-world dataset consisting of over 4.7M audit log entries, and 6) a performance comparison of our method of generating ABAC policies with another algorithm for generating RBAC policies. This work demonstrates the effectiveness of our methodology for implementing least privilege and generating ABAC policies from complex environments. It also demonstrates that with proper design, an ABAC approach is able to produce policies with less over-privilege and under-privilege even based on less data than an RBAC approach.

The rest of this paper is organized as follows. Section 2 provides background information on ABAC and rule mining methods, and reviews related work. Section 3 formally defines the $PEMP_{ABAC}$ problem and metrics for evaluating policies. Section 4 describes our algorithms for addressing the $PEMP_{ABAC}$ problem. Section 5 analyzes the results of applying our algorithms to a real-world dataset. Section 6 concludes our work.

2 BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Attribute Based Access Control (ABAC). ABAC is an access control model where a subject's requests to perform operations on objects are granted or denied based on "the assigned attributes of the subject, the assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions" [8]. Attributes are any properties of the subjects, objects, and environment encoded as *name:value* pairs. Subjects may be persons or non-person entities, objects are system resources, operations are functions executed upon objects at the request of subjects, while environment conditions are characteristics of the context in which access requests occur and are independent of subjects and objects [8]. ABAC's flexibility allows it to implement traditional access control models such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC).

RBAC [17] is flexible. It has been widely deployed and used for more than two decades. However, as access control needs have become more complex and applied to more diverse domains, organizations have found that RBAC does not provide sufficient granularity, becomes difficult to manage, or does not support their information sharing needs. Organizations facing these challenges may address them using an ABAC based system. Consider the case of restricting access for performing a database backup to a specific timeframe and IP address range. Such constraints can be easily expressed using ABAC attributes, but cannot be expressed using only the user, operation, and object semantics of the RBAC model.

2.1.2 Rule Mining Methods. Frequent itemset mining is a popular method for identifying patterns with applications in many diverse fields [7]. The frequent itemset problem is defined as follows: given a transaction database DB and a *minimum support threshold* ϵ, find the complete set of frequent patterns. The set of items is $I = \{a_1, ..., a_n\}$ and a transaction database is $DB = \langle T_1, ..., T_m \rangle$, where $T_i (i \in [1...m])$ is a transaction which contains a set of items in I. The *support* of a pattern A (where A is a set of items) is the fraction of transactions containing A in the DB: $support(A) = \frac{|T_i \in DB | A \subseteq T_i|}{|DB|}$. A pattern is *frequent* if A's support is $>= \epsilon$ (which is the minimum support threshold) [6]. The output of frequent itemset mining is many subsets of items that occurred within the transaction database DB. In the context of creating access control policies, there is a clear translation of frequent itemsets into ABAC rules, and generating candidate rules from these frequent itemsets is a key component of our rule mining algorithm (Section 4.1).

2.2 Related Work

We group related work into two categories: those that deal with generating least privilege RBAC policies, and those that address the problems of modifying or creating ABAC policies of minimal size. To the best of our knowledge, our work is the first to address the problem of automatically creating least privilege ABAC policies.

2.2.1 Least Privilege Policy Generation. In [16], the authors defined the Privilege Error Minimization Problem (PEMP) for RBAC, and designed naive, unsupervised, and supervised learning algorithms to minimize privilege assignment errors in RBAC policies. Another important work in generating least privilege policies is [12], which used Latent Dirichlet Allocation (LDA) to create least privilege RBAC policies from logs of version control software. This work used user attribute information in the mining process although the resulting policies were RBAC policies. The authors introduced the $\lambda - Distance$ metric for evaluating candidate rules [12]. This metric adds the total number of under-assignments to the total number of over-assignments with λ acting as a weighting factor on the over-assignments to specify how much it values over-privilege vs. under-privilege for a particular application.

In comparison with these two works, we formally define PEMP for ABAC in this paper. Moreover, we present new algorithms, new metrics, and new optimization methods that are all necessary in dealing with the much larger ABAC privilege space to properly implement least privilege in ABAC policies.

2.2.2 ABAC Policy Mining. One early work on applying association rule mining to ABAC policies was [5], which used the Apriori algorithm [1] to detect statistical patterns from access logs of lab doors in a research lab. The dataset consisted of 25 physical doors and 29 users who used a smart-phone application and Bluetooth to open the doors. The authors used the output of the mining algorithm to identify policy misconfigurations by comparing mined rules with existing rules.

In [9], the authors presented a tool named Rhapsody which also uses the Apriori algorithm. Rhapsody seeks to create ABAC policies of minimal over-privilege by mining logs. However, it does not provide a weighting method for balancing between under-privilege and over-privilege, nor does it consider large and complex privilege spaces. Rhapsody uses a simplified model of attributes with Users and Permissions only. While Rhapsody is designed to operate on "sparse" audit logs where only a small amount ($\leq$ 10%) of all possible log entries are likely to occur in the mined logs, our work is designed to operate on logs several orders of magnitude more sparse than those of Rhapsody by using optimization techniques described

in Section 4.3. In addition, the run time of Rhapsody grows exponentially with the maximum number of rules a request may satisfy, limiting the number of attributes that can be considered to "less than 20" [9], which would prevent a direct comparison between their approach and ours using our dataset of over 1,700 attributes.

Xu and Stoller [21] presented an algorithm to create ABAC policies that cover all the entries found in an audit log while also minimizing the size of the overall policy through a process of merging and simplifying candidate rules until all the given privilege tuples are covered. Their evaluation metric is an ABAC version of Weighted Structural Complexity (WSC), which was originally presented in [11] as a measure for the size of RBAC policies. Their algorithm uses a simplified ABAC model, and calculates coverage based on *user-permission tuples*, where a tuple $\langle u, o, r \rangle$ contains a user, operation, and resource only, instead of considering all the valid attribute combinations in the privilege space. This reduces the computational complexity of mining and evaluating rules, but presents a problem for accurately evaluating ABAC policies because such a tuple may be either allowed or denied unless considering the attributes of user, operation, and resource at the request time.

In comparison with these works, we use a model including Users, Operations, Resources, and Environment attributes; we address the challenges in systems with large and complex privilege spaces; we measure under- and over-privilege of policies in our evaluation instead of other metrics such as policy size and complexity; we use out-of-sample validation to capture the performance of mined policies over time given they were put into operation. While minimizing complexity (evaluated by WSC) is desirable in that it makes policies easier to maintain by administrators, we see it as less important than least privilege performance over time. This is especially true when using automated methods to build policies where less administrator involvement is necessary. Methods for minimizing ABAC policy complexity are complementary to our work as once least privilege policies are identified, then methods for minimizing policy complexity can be applied.

3 PROBLEM DEFINITION AND METRICS

This paper addresses the problem of minimizing privilege assignment errors in ABAC policies. Access control can be viewed as a prediction problem. The statements of a policy are predictions about which entities should be granted privileges to perform specific operations upon the specific resources necessary to perform their jobs. The goal of this work is to automatically generate policies that are accurate access control predictions. To help clarify the specific problem this paper addresses, we formally define it as the ABAC Privilege Error Minimization Problem ($PEMP_{ABAC}$) in this section. We also define metrics to be used in evaluating the performance of proposed solutions.

3.1 Problem Definition

Our problem definition is based on the Privilege Error Minimization Problem (PEMP) originally defined in [16] for creating least privilege RBAC policies which consisted of users, operations, and objects. Like the original PEMP, our problem seeks to minimize the under- and over-privilege assignment errors in policies and uses the notions of observation and operation periods for evaluation. However, users, operations, and resources are only some of the attributes available when creating ABAC policies; therefore, a unique problem definition in the ABAC privilege space is needed.

The size of an ABAC privilege space is determined by the attributes and values of valid ABAC policies. $\mathbb{A}$ is the set of valid attributes which can be used in policies. As in related works [5, 14, 21], we assume all attributes and values existing in the logs can also be used in policies. Each individual attribute $a_i \in \mathbb{A}$ has a set of atomic values $\mathbb{V}_i$ which are valid for that attribute. All values for an attribute are the attribute's range $Range(a_i) = \mathbb{V}_i$. The Cartesian product of all possible *attribute:value* combinations is $\xi = \mathbb{V}_1 \times ... \times \mathbb{V}_n = \{(v_1, ..., v_n) | v_i \in \mathbb{V}_i \text{ for every } i \in \{1, ..., n\}\}$. However, some *attribute:value* pairs are not valid when present in combination with other *attribute:value* pairs because of dependencies between them. For example, some operations are only valid on certain resource types so combinations such as *operation:DeleteUser* and *resourceType:File* are not valid. The valid privilege universe ξ' is the set of all possible *attribute:value* combinations when considering the dependency relationships between all attributes and values.

Any measure of security policy accuracy must also take *time* into account because the amount of risk from over-privilege accumulates over time. Over-privilege carries the risk that an unnecessary privilege will be misused, and this risk increases the longer the over-privilege exists. To capture risk across a specified time period, we define the *Operation Period* (*OPP*) as the time period during which security policies are evaluated against user operations. With the valid privilege universe ξ' and the operation period *OPP* defined, we now define the ABAC version of the Privilege Error Minimization Problem $PEMP_{ABAC}$ (Definition 1).

Definition 1. $PEMP_{ABAC}$: ABAC Privilege Error Minimization Problem. *Given the universe of all valid attribute:value combinations ξ', find the set of attribute:value constraints that minimizes the over-privilege and under-privilege errors for a given operation period OPP.*

3.2 Evaluation Metrics

We use terminology from statistical hypothesis testing for evaluating the effectiveness in addressing the $PEMP_{ABAC}$. We first present our method for scoring individual predictions, and then our method for splitting up the dataset and evaluating the performance over multiple time periods.

3.2.1 Scoring Individual Predictions. Policy evaluation for a given operation period is a two-class classification problem where every possible event in the ABAC privilege space falls into one of two possible classes: grant or deny. By applying the policies generated from the observation period data to the privileges exercised in the operation period, we can categorize each prediction into one of four outcomes:

- True Positive (TP): a privilege that was granted in the predicted policy and exercised during the OPP.
- True Negative (TN): a privilege that was denied in the predicted policy and not exercised during the OPP.
- False Positive (FP): a privilege that was granted in the predicted policy but not exercised during the OPP.

- False Negative (FN): a privilege that was denied in the predicted policy but attempted to be exercised during the OPP.

Using the above outcomes we then calculate True Positive Rate (**TPR**) also known as **Recall** and False Positive Rate (**FPR**) as shown in Formulas 1 and 2, respectively:

$$TPR = \frac{TP}{(TP + FN)} \tag{1}$$

$$FPR = \frac{FP}{(FP + TN)} \tag{2}$$

RBAC mining in [16] used metrics based on TPR and Precision, while our ABAC mining has to use TPR and FPR instead. Precision ($\frac{TP}{TP+FP}$) is suitable when considering the users and operations because the universe of possible grants is roughly on the same order of magnitude as the number of unique log events. When dealing with the ABAC universe, the number of possible unique *attribute:value* combinations is likely to be many orders of magnitude greater than the number of events in the operational logs. Precision is not a suitable metric for use in mining ABAC policies from logs because it uses one term (TP) which is driven primarily by the number of entries in the log, and another term (FP) which is driven by the size of the privilege universe. On the other hand, both terms in the TPR (TP and FN) are log derived, and both terms in FPR (FP and TN) are policy derived metrics.

TPR and FPR are the metrics used to evaluate a policy in terms of under-privilege and over-privilege, respectively. If all privileges exercised in the *OPP* were granted, there was no under-privilege for the policy being evaluated so $FN = 0$, and $TPR = 1$. As the number of erroneously denied privileges (FNs) grows, $TPR \rightarrow 0$, thus TPR represents under-privilege. If all privileges granted by the policy were exercised during the *OPP*, there was no over-privilege for the policy being evaluated so $FP = 0$ and $FPR = 0$. As the number of erroneously granted privileges (FPs) grows, $FPR \rightarrow 1$, thus FPR represents over-privilege.

3.2.2 Scoring Policies Across Multiple Time Periods. To score policies across multiple time periods, we use out-of-time validation [10], a temporal form out-of-sample validation. In out-of-sample validation, a set of data is used to train an algorithm (training set) and a separate set of non-overlapping data is used to test the performance of the trained algorithm (test set). In our evaluation, the training and test sets are contiguous, and the test time period immediately follows the training time period. The training set is referred to as the *Observation Period* (*OBP*), while the test set is the Operation Period (*OPP*) defined previously in Section 3.1. It is important to note that this method preserves the temporal interdependencies between actions. For example, if an employee moves to a new position within the organization, one would expect the privileges mined for that employee in the future time periods would be very different from those mined in the past time periods. Methods such as k-fold cross validation which randomly partition a dataset (as used in [12] for evaluating policies) do not account for these temporal interdependencies. When charting metrics for multiple time periods, we use the average of all individual scores. This gives equal weight to each operation period score.

3.2.3 Scoring Infinite Possible Resource Identifiers. Quantifying the number of resources allowed or denied by a policy implies that there is a known value for the number of possible resources in the system. This presents a challenge to any least-privilege scoring approach, and not just to the ABAC model or our methodology. While every system has finite limits on the resource identifier length and number of resources, these can be so numerous that we consider them as too large to quantify and treat them as being infinite. For example, consider the number of possible file names in an ext4 file system with up to 255 bytes for a file name, $2^{8^{255}}$ possible distinct file names exist excluding the path [13].

Instead of counting all possible resource identifiers, we use the resource identifiers existing in the *OBP* and *OPP* for our policy scoring calculations. This approach presents several advantages over other possible approaches such as using all values in a dataset, or introspecting an environment for the resource identifiers (which would be prohibitively time consuming for our work). Only the recently used resources are counted, giving them greater importance, and all necessary data is available in the audit logs. This also implies that the valid privilege space ξ' may vary in size between scoring periods depending on the resource identifiers present.

4 METHODOLOGY

This section presents our rule mining algorithm for addressing the $PEMP_{ABAC}$ problem, policy scoring algorithm for evaluating the policies across multiple operation periods, and optimization methods for processing large ABAC privilege spaces. Before going into the details, we first describe the overall workflow of our approach for mining least privilege ABAC policies as shown in Figure 1.

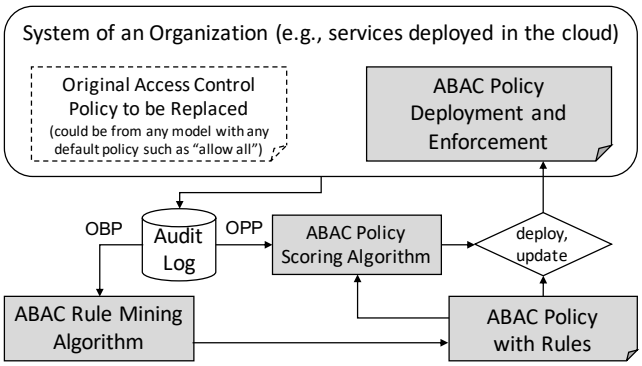

Figure 1: Overall Workflow of our Approach

A system under operation or in testing will continuously generate audit log events for access requests (either allowed or denied). The events of any chosen Observation Period (OBP) will be the input to the ABAC Rule Mining Algorithm, which generates an ABAC policy with a set of rules as the output. This policy is then scored against a subsequent Operation Period (OPP) of audit log events to evaluate its performance as if it were put into operation. If this policy can better balance or minimize under- and over-privilege, it would be deployed to the system. This is an iterative and continuous process, so that a newly deployed policy can also be adaptive to user behavior and situation changes over time.

This workflow can be easily bootstrapped as long as the system generates audit log events. It does not depend on the model (e.g.,

RBAC or ABAC), content, and even the existence of an original access control policy. For example, an administrator may choose to begin with a simple "allow all" style of policy based on whatever model. Our approach will then periodically mine a new ABAC policy, evaluate its performance (quantitatively regarding the level of under- and over-privilege if the policy were deployed), and deploy or update an improved policy to the system either automatically or with the confirmation from the administrator.

4.1 Rule Mining

Our rule mining algorithm operates similarly to the mining algorithms presented in [12, 21] in that it considers the set of uncovered log entries and iteratively generates many candidate rules, scores them, and selects the best scoring rule for the next iteration until all of the given log events are covered by the set of generated rules. A critical component of this approach is the metric used to evaluate candidate rules. Before describing the algorithm design, we will first detail the metric used for evaluating candidate rules generated during the mining process. We propose a candidate scoring metric termed C_{score} using the following definitions.

- c is an ABAC constraint specified as a *attribute:value* pair, or a key with a set of values *key:*$\{values\}$. Values must be discrete, so continuous ones should be binned to be used by the mining algorithm. r is a rule consisting of one or more constraints. p is a policy consisting of one or more rules.
- $\mathbb{L}$ is the complete set of log entries for the dataset, $\mathbb{L}_{OBP}$ is the set of logs in the observation period OBP; $\mathbb{L}_{OBP} \subseteq \mathbb{L}$.
- $\mathbb{L}_{OBP}(C)$ is the set of log entries which meet (i.e., are "covered by") the constraints in a set C that can be specified by the use of a rule r or policy p; $\mathbb{L}_{OBP}(C) \subseteq \mathbb{L}_{OBP}$.
- ξ' is the privilege universe of valid log events as defined previously in Section 3.1.

The *CoverageRate* (Formula 3) is the ratio of all logs in the observation period covered by a candidate rule r but not already covered by other rules in the policy p ($|\mathbb{L}_{OBP}(r) \setminus \mathbb{L}_{OBP}(p)|$) to the remaining number of log entries not covered by any rules in the policy ($|\mathbb{L}_{OBP} \setminus \mathbb{L}_{OBP}(p)|$). A candidate rule that covers more log entries is considered higher quality than a rule that covers fewer log entries. The numerator of the *OverPrivilegeRate* (Formula 4) finds the number of valid *attribute:value* combinations in the privilege universe covered by a rule ($\xi'(r)$) minus the set $\mathbb{L}_{OBP}(r) \setminus \mathbb{L}_{OBP}(p)$, resulting the total number of over-assignments for rule r. The total over-assignments are then normalized upon the total number of combinations in the valid privilege universe $|\xi'|$. A candidate rule which has fewer over-assignments is considered higher quality than a rule that has more over-assignments.

$$CoverageRate(r, p, \mathbb{L}_{OBP}) = \frac{|\mathbb{L}_{OBP}(r) \setminus \mathbb{L}_{OBP}(p)|}{|\mathbb{L}_{OBP} \setminus \mathbb{L}_{OBP}(p)|} \tag{3}$$

$$OverPrivilegeRate(r, p, \mathbb{L}_{OBP}, \xi') = \frac{|\xi'(r) \setminus (\mathbb{L}_{OBP}(r) \setminus \mathbb{L}_{OBP}(p))|}{|\xi'|} \tag{4}$$

The candidate score C_{score} (Formula 5) is then the ω weighted addition of the *CoverageRate* and the complement of the *OverPrivilegeRate*. By normalizing the under-assignments using the number of log entries and the over-assignments using the size of the valid privilege universe, the effect of varying the weight ω in C_{score} is more predictable and better performance can be achieved when compared to the $\lambda{-}Distance$ metric which also uses a variable weighting between over-assignments and under-assignments but does not normalize these values (see Section 5.2 for the C_{score} vs. $\lambda{-}Distance$ comparison details).

$$\begin{aligned} C_{score}(r, p, \mathbb{L}_{OBP}, \xi', \omega) = CoverageRate(r, p, \mathbb{L}_{OBP}) + \\ \omega \times (1 - OverPrivilegeRate(r, p, \mathbb{L}_{OBP}, \xi')) \end{aligned} \tag{5}$$

Our algorithm for mining an ABAC policy from the logs of a given observation period is presented in Algorithm 1. Note that we use arithmetic operators $=, +, -$ when describing integer operations, and set operators $\leftarrow, \cup, \setminus, \in, |\ldots|$ when describing set operations. As mentioned previously, the algorithm iteratively generates candidate rules from the set of uncovered logs. To avoid confusion between the original set of log entries for the observation period $\mathbb{L}_{OBP}$ and the current set of uncovered log entries which is updated for each iteration of the algorithm, we copy $\mathbb{L}_{OBP}$ to $\mathbb{L}_{uncov}$ at line 2. The *FP-growth* algorithm [6] is used to mine frequent itemsets from the set of uncovered observation period log entries (line 4). The itemsets returned by the *FP-growth* algorithm are sets of *attribute:value* statements, and each of these itemsets is used to create a candidate rule which is then scored using the C_{score} metric (lines 6-12). After all candidates are scored, the highest scoring rule is selected and added to the policy, then all log entries covered by that rule are removed from the set of uncovered log entries (lines 13-15). This process continues until all log entries are covered (lines 3-16).

4.2 Policy Scoring

Once the observation period logs have been mined to create a policy, that policy is scored using the events that took place during the operation period immediately following the mined observation period as described in Algorithm 2. Each event during the operation period is evaluated against the mined policy (lines 3-10). Events allowed by the policy are TPs, while events denied by the policy are FNs. A unique combination of *attribute:value* pair may occur multiple times within the same time period. The TPs and FNs are both values based on the number of times an event occurs in the log. The set of unique events that were exercised in the operation period and granted by the policy is also maintained (line 6) in order to calculate the FPs later (line 15). By counting each TP and FN instead of unique occurrences, the resulting TPR is frequency weighted. Events that occur more frequently in the operation period have a greater impact on TPR than those events that occur less frequently.

While the TPs, FNs, and resulting TPR are based on the frequency weighted count of events present in the log, the FPs, TNs and resulting FPR cannot be frequency weighted because each unique valid event of the privilege universe is either granted or denied only once by the policy. To obtain these values (FP, TN, FPR), we first determine how many unique events out of the valid privilege space are granted by the policy (lines 11-14). It is important to note that enumerating the entire privilege space and testing every

Algorithm 1: Rule Mining Algorithm

Input: $\mathbb{L}_{OBP}$ The set of log entries representing user actions during the observation period OBP.
Input: ω under- vs. over-privilege weighting variable.
Input: ϵ Threshold value of minimum itemset frequency.
Input: ξ‘ The set of all *attribute:value* combinations in the valid privilege universe.
Output: $policy$ The policy with a set of ABAC rules to be applied during the operation period OPP.

1 $policy \leftarrow \emptyset$;
2 $\mathbb{L}_{uncov} \leftarrow \mathbb{L}_{OBP}$;
3 **while** $|\mathbb{L}_{uncov}| > 0$ **do**
4 $\quad itemsets \leftarrow FP\text{-}growth.frequentItemsets(\mathbb{L}_{uncov}, \epsilon)$;
5 $\quad candidateRules \leftarrow \emptyset$;
6 $\quad$ **for** $itemset \in itemsets$ **do**
7 $\quad\quad rule = createRule(itemset)$;
8 $\quad\quad coverageRate = \frac{|\mathbb{L}_{uncov}(rule)|}{|\mathbb{L}_{uncov}|}$;
9 $\quad\quad overAssignmentRate = \frac{|\xi\text{‘}(rule)| - |\mathbb{L}_{uncov}(rule)|}{|\xi\text{‘}|}$;
10 $\quad\quad rule.C_{score} = coverageRate + \omega \times (1 - overAssignmentRate)$;
11 $\quad\quad candidateRules \leftarrow candidateRules \cup rule$;
12 $\quad$ **end**
13 $\quad bestRule = sortDescending(candidateRules, C_{score})[0]$;
14 $\quad policy \leftarrow policy \cup bestRule$;
15 $\quad \mathbb{L}_{uncov} \leftarrow \mathbb{L}_{uncov} \setminus \mathbb{L}_{uncov}(bestRule)$;
16 **end**
17 **return** $policy$

Algorithm 2: Policy Scoring Algorithm

Input: $\mathbb{L}_{OPP}$ The set of log entries representing user actions during the operation period OPP.
Input: ξ‘ The set of all *attribute:value* combinations in the valid privilege universe.
Input: $policy$ The policy with a set of ABAC rules to be applied during the operation period OPP.
Output: TPR, FPR True and false positive rates of the $policy$ evaluated on the operation period OPP.

1 $TP = FN = 0$;
2 $exercisedGrantedEvents \leftarrow \emptyset$;
3 **for** $event \in \mathbb{L}_{OPP}$ **do**
4 $\quad$ **if** $policyAllowsEvent(policy, event)$ **then**
5 $\quad\quad TP = TP + 1$;
6 $\quad\quad exercisedGrantedEvents \leftarrow exercisedGrantedEvents \cup event$;
7 $\quad$ **else**
8 $\quad\quad FN = FN + 1$;
9 $\quad$ **end**
10 **end**
11 $eventsAllowedByPolicy \leftarrow \emptyset$;
12 **for** $r \in policy$ **do**
13 $\quad eventsAllowedByPolicy \leftarrow eventsAllowedByPolicy \cup \xi\text{‘}(rule)$;
14 **end**
15 $FP = |eventsAllowedByPolicy \setminus exercisedGrantedEvents|$;
16 $TN = |privUniverse| - (TP + FN + FP)$;
17 **if** $TP + FN == 0$ **then**
18 $\quad TPR = 1$;
19 **else**
20 $\quad TPR = TP/(TP + FN)$;
21 **end**
22 $FPR = FP/(FP + TN)$;
23 **return** TPR, FPR

valid event against the policy would be much more computationally intensive than our approach, which is to use information about the valid privilege space to enumerate only the valid events allowed by each rule. Most mined rules only allow a small percentage of the privilege space except in cases of extreme ω values.

Once the set of all the unique events allowed by a policy has been enumerated, we remove the set of unique events which occurred and were granted during the operation period to obtain the number of total FP events for the policy (line 15). At this point we have obtained the unique sets of TPs, FNs, and FPs, so any remaining privilege in the valid privilege universe not in these sets must be a TN (line 16). TPR and FPR are then calculated with the caveat that in the case where no privileges were exercised during the operation period, we set $TPR = 1$ because there could not be any instances of under-privilege (lines 18-22). The *policyAllowsEvent()* function is self-explanatory and its trivial implementation is omitted.

4.3 Optimizations for Large Privilege Spaces

Dealing with the large number of possible *attributes:value* combinations that may comprise an ABAC privilege space is a significant challenge compared to the simpler RBAC privilege space. Using all attributes and values present in logs may make the privilege universe computationally impractical to process, but discarding too many or important attributes may result in less secure policies. We address these issues and make large ABAC privilege spaces manageable by using feature selection and partitioning methods.

4.3.1 Preprocessing and Feature Selection. Intuitively, attributes which occur infrequently in the logs or have highly unique values are poor candidates for use in creating ABAC policies. There is less data available to mine meaningful patterns from uncommon attributes. Also, rules created with uncommon attributes are less useful in access control decisions because future access requests are unlikely to use these attributes as well. Using attributes with highly unique values (an attribute value is never or rarely duplicated across log entries) is likely to result in over-fitting for the correspondingly created rules. We therefore preprocess our dataset to select and bin the most useful attributes as follows:

(1) Remove unique and redundant attributes using $Uniqueness$ where $Uniqueness = \frac{|UniqueValues|}{AttributeOccurrences}$.
(2) Remove redundant correlated attributes.
(3) Sort attributes by $Frequency = \frac{AttributeOccurrences}{TotalLogEntries}$. Select attributes above a frequency threshold, θ.
(4) Sort remaining values by $Uniqueness$. High $Uniqueness$ values are candidates for binning or removal.

Our full AWS dataset contained 1,748 distinct attributes (see Section 5.1 for dataset description). In step (1), $|UniqueValues|$ is obtained by calculating the size of the set of all unique values for

every attribute. Set attributes with $Uniqueness \approx 1.0$ nearly always have unique values, and $Uniqueness \approx 0.0$ implies the attribute values are nearly always the same. Resource identifiers are given an exception to the uniqueness test in this step as they are expected to have high uniqueness. For our dataset, we identified and removed two always unique attributes, *eventID* and *requestID*, and one attribute that always had the same value *accountId*. We confirmed that these attributes would always meet the uniqueness criteria with the AWS documentation.

Applying step (2), we identified three distinct attributes for the user name with a 1:1 correlation and removed two of them. The reason for this is if three given values are always 1:1 correlated, the data mining algorithm gains nothing by having all of them - two of them are redundant and can be removed without loss of discriminating power.

For step (3), we selected two thresholds, $\theta = 0.1$ and $\theta = 0.005$, to build two datasets for experiments, and we term the privilege universes built using these thresholds $\xi'_{0.1}$ and $\xi'_{0.005}$, respectively. Figure 2 charts the rank of the top 50 most common attributes after our feature selection process was complete. The attribute frequency follows the common power law distribution with a "long tail"; the remaining attributes not charted here occurred in less than 0.2% of the log entries. $\xi'_{0.1}$ and $\xi'_{0.005}$ correspond to top 15 and 40 attributes ranked by frequency, respectively. Based on the chart, these two frequency values were chosen as cut off points after which the amount of information gained becomes more negligible.

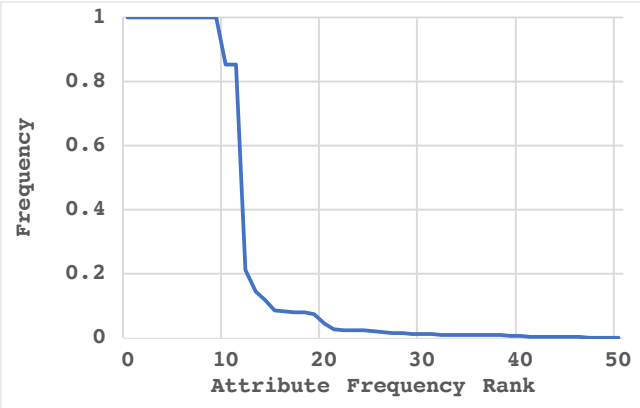

Figure 2: Top 50 Attributes Ranked by Frequency

In step (4), some of the remaining attributes still have fairly high *Uniqueness* values which are difficult to mine meaningful rules from. In our dataset, some of these attributes such as checksum values are not relevant to creating security policies and can be discarded. Three attributes, *sourceIPAddress, userAgent, and eventName*, can benefit from binning into a smaller subset of values. The *sourceIPAddress* is an IPv4 address with over 4 billion possible values. After consulting with the system administrators of the dataset provider, we found that it was unlikely they would use rules based on the raw IP addresses since users will change IPs frequently. Instead, they preferred to derive the geographical location from the IP addresses, so IPs were binned by U.S. states and each country the organization's users may log in from. The *userAgent* attribute is the AWS Command Line Interface (CLI), Software Development Kit (SDK), or web browser version used when making a request. This field benefits from binning as users are likely to perform similar requests from a web browser, but they may upgrade their browser version regularly. Without binning the many different browser versions into a single group, a mining algorithm would not effectively learn user patterns. Again, the dataset provider agreed that the raw value was too granular to use, so the *userAgent* attribute was binned into 10 buckets. The *eventName* attribute is the name of the operation. This attribute can be effectively binned because each *eventName* is associated with one *eventSource*, which is the AWS service name associated with the operation.

4.3.2 Mining Algorithm Optimizations. The resulting ABAC privilege space may still be quite large even for a modest dataset after applying the feature selection and binning methods as just described. We further apply partitioning techniques to split up the privilege space in the policy mining process. Partitioning techniques (as used in databases to split large tables into smaller ones) will help reduce the memory footprint of our algorithms and improve efficiency by performing operations in parallel across multiple processors.

The rule mining algorithm (Algorithm 1) uses partitioning to improve the run time and space efficiency for storing and searching the privilege universe ξ'. The total number of valid combinations in ξ' was on the order of billions for some of our experiments, but Algorithm 1 only needs to determine the number of privileges covered by a rule and needs not to enumerate or store all possible privilege combinations in memory. This is a subtle but important difference because it means we can calculate the number of valid privilege combinations by splitting ξ' into smaller sets of independent partitions. The total number of valid privilege combinations covered by a rule is the product of the number of valid privilege combinations covered by each partition, i.e., $|\xi'(r)| = |P_1(r)| \times \ldots \times |P_n(r)|$, where the attributes of each partition P_i are independent of the attributes in all other partitions.

To create these partitions, the AWS documentation was used to identify dependencies between attributes in our dataset. Next, a simple depth first search was used to identify connected components of interdependent attributes. The valid *attribute:value* combinations for all attributes in each connected component were then enumerated and stored into one inverted index for each partition. Finding the number of valid privilege combinations covered by a rule in a partition ($|P_n(r)|$) is accomplished by searching the inverted index using the rule's *attribute:value* constraints as search terms. As a result of this partitioning, our queries were performed against three indexes on the order of thousands to hundreds of thousands of documents vs. a single index that would have been on the order of hundreds of millions to billions of documents if such a partitioning scheme were not in use.

For our dataset, a depth first search identified one connected component of all user attributes, and another connected component of operations and resources. Operations and resources were connected because most operations are specific to a single or set of resource types. We grouped all other attributes that were independent of users and operations into a third component which included environment attributes such as the *sourceIPAddress* and *userAgent*. Although this grouping of attributes by components was obtained

from processing our specific dataset, it is reasonable to assume that user attributes are independent of the valid operation and resource attribute combinations in other datasets as well. This is also consistent with the NIST ABAC guide which defines environment conditions as being independent of subjects and objects [8].

Due to the large number of candidate rules generated by the $FP{-}growth$ algorithm, scoring candidate rules is the most computationally intensive part of Algorithm 1 in our experiments (except for those with fairly large ϵ values which generate few candidates). The search against the inverted index is also parallelized to improve performance.

4.3.3 Scoring Algorithm Optimizations. To improve the run time performance of the policy scoring algorithm (Algorithm 2) and enable it to deal with a privilege space larger than the available memory, we again employ partitioning and parallelization methods. As mentioned in 4.2, Algorithm 2 must enumerate the set of all privilege combinations covered by a rule in order to identify the total unique number of privilege combinations covered by a policy. If extreme values for ω are chosen, it is possible for Algorithm 1 to generate rules with a large number of over-privileges, possibly the entire privilege space. Therefore, Algorithm 2 must be able to deal with the possibility that it will have to enumerate all privilege combinations of ξ‘, although again, this only happens for extreme values of ω and is only for the out-of-sample validation in policy scoring rather than in rule mining.

To deal with the possible need to enumerate a large portion or even all of the privilege space, we partitioned ξ‘ along two attributes so that the values of those attributes are placed into separate partitions. As with any partitioning, choosing a key that nearly equally splits the universe of possible values is important. In our experiments, we chose to partition the ξ‘ space along the attributes associated with the operation name and user name. The overall correctness of the algorithm is independent of the partition keys used, and 1 to n partitions may be used for each attribute depending on the size of the privilege space and available memory.

Each of these partitions is operated on in parallel when evaluating each rule of the policy. Unique hashes of the enumerated events are used in order to deduplicate events which may be generated by more than one rule. This partitioning and parallelization takes place within lines 11-14 of Algorithm 2. We describe these optimizations here because they are useful in speeding up and scaling the algorithm when dealing with a large number of *attribute:value* pairs, but we omit them from the pseudo-code of Algorithm 2 to simplify its presentation.

5 RESULTS

We use the Receiver Operating Characteristic (ROC) curve to compare the performance of various algorithms and parameters. The ROC curve charts the trade-off between the TPR and FPR of a binary classifier, with the ideal performance having a TPR value of one and a FPR value of zero. Our charts also include the Area Under the Curve (AUC), which measures the area underneath the ROC curve and provides a single quantitative score that incorporates both *FPR* and *TPR* as the weighting metrics being varied. The higher the *AUC* score the better the classification performance.

First, we describe our dataset used for the experiments. Next we present experimental results and analysis to justify our candidate evaluation metric C_{score}, including a comparison of several methods for normalizing the $CoverageRate$ variable. Then we examine the effect of varying two adjustable input variables to the mining algorithm: the length of the observation period ($|\mathbb{L}_{OBP}|$) and the minimum support threshold (ϵ). Finally, we compare the performance of our ABAC algorithm with that of an RBAC algorithm.

All charts presented in this section are based off of mining and scoring the entire dataset $|\mathbb{L}|$ using the Algorithms 1 and 2. The exact number of runs varies based on the observation period size $|\mathbb{L}_{OBP}|$ used and equals to $|\mathbb{L}| - |\mathbb{L}_{OBP}| + 1$.

5.1 Dataset Description

We examine the performance of our ABAC policy generation algorithm on a real-world dataset. Our dataset was provided by a Software As A Service (SaaS) company that uses 77 different AWS services [4] with an “allow all” style of RBAC policy. It consists of 4.7M user-generated AWS CloudTrail audit events [2] representing 16 months of audit data starting from March 2017 for 38 users. CloudTrail logs the events of all AWS account activities, including actions taken through the AWS Management Console, AWS SDKs, command line tools, and other AWS services; it performs the audit logging for all services at the platform and infrastructure layers which are also the layers that AWS IAM (Identity and Access Management) enforces access controls for [2]. Audit events are logged by CloudTrail in JSON format, and can be easily parsed by using any JSON library. Two audit event examples from the AWS website [3] are provided in Appendix A. Note that we used user-generated audit events only, filtering out those events generated by non-person entities. Events generated by non-person entities were very consistent, and it is easy to derive very low under- and over-privilege security policies for them directly from audit logs without using advanced methods. Minimizing the privilege assignment errors for human users is much more challenging so we chose to focus on human generated log events only.

The high degree of variability in user behavior is shown by the statistics in Table 1 based on the first month, last month, and total 16 months of data. *Users* is the number of active users during that time period. *Unique Services Avg.* is the average number of unique services used by active users. *Unique Actions Avg.* is the average number of unique actions exercised by active users, and $\sum$ *Action Avg.* is the average of the total actions exercised by active users. The standard deviation is provided for Unique Services, Unique Actions, and $\sum$ Actions metrics to understand the variation between individual users. For example, looking at both the Unique and $\sum$ Actions, we observe that their standard deviation is higher than the average for all time periods, indicating a high degree of variation between the number of actions that users exercise.

From our initial dataset, we derive two privilege universes using our feature selection methodology (Section 4.3.1). $\xi'_{0.1}$ used 15 attributes and consisted of 510M unique *attribute:value* combinations. $\xi'_{0.005}$ used 40 attributes, 25 of which were resource identifiers so the universe size varied between 1.5B and 8.6B unique *attribute:value* combinations depending on the number of resources used during the *OBP* and *OPP* periods. All the experiments in this

Table 1: 16-Month Total Usage of our Dataset

Metric	First Month	Last Month	16 Months
Users	17	26	38
Unique Services Avg.	12.94	12.11	22.66
Unique Services StdDev.	10.16	9.98	16.70
Unique Actions Avg.	65.76	62.92	168.34
Unique Actions StdDev.	76.11	73.30	178.52
$\sum$ Actions Avg.	9138.35	9664.19	123659.82
$\sum$ Actions StdDev.	20279.87	14124.15	235915.45

section use $\xi`_{0.1}$ except for Section 5.4 which uses $\xi`_{0.005}$. Our rule mining algorithm (Algorithm 1) currently saves a generated policy in JSON format. Appendix B provides an example rule of a policy.

5.2 C_{score} Analysis

We consider three criteria in the design and evaluation of the C_{score} metric for selecting a single rule from many candidate rules generated by the *FP−growth* algorithm during each iteration of our rule mining algorithm. Criterion *C*1:*AUC* is the Area Under the ROC Curve. Criterion *C*2:*Smoothness* means that *TPR* values should increase monotonically as the *FPR* increases. Criterion *C*3:*Interpretability* means that the effect of changing the weighting variable should be predictable and easy to understand by an administrator who uses the metric in a policy mining algorithm.

5.2.1 Evaluating Candidate Scoring Metrics. Our candidate scoring metric C_{score} is presented in Section 4.1, *λ−Distance* is presented in [12], and *Qrul* is presented in [21]. All these metrics use the number of over-assignments and number of log entries covered with a weighting variable for adjusting the importance between over-assignments and coverage in their scoring of candidates. However, these metrics differ in how they normalize these numbers (if at all) and how they implement the weighting. The results of varying the over-assignment weightings for these candidate evaluation methods are shown in Figure 3.

Four distinct versions of the *Qrul* metric are included in Figure 3. In [21], the authors also described *QrulFreq*, a frequency weighted variant of *Qrul* which should be a fairer comparison with our frequency weighted policy scoring algorithm (Algorithm 2). The authors of [20] provided their source code on their website, and the scoring algorithms implemented in the source code for *Qrul* and *QrulFreq* are slightly different from those presented in the paper. Instead of using the number of privileges covered by a rule out of the entire privilege universe ($[\![p]\!]$) as the denominator for the over-assignments side of the metric, the implemented metrics instead use the number of privileges covered by a rule out of the log entries not covered by other rules already in the policy ($|[\![p]\!] \cap UP|$). These "as-implemented" metrics, *QrulImpl* and *QrulFreqImpl*, perform more favorably than their counterparts so we include them in our comparison along with the versions documented in [21].

All the examined metrics performed relatively well with high *AUC* values, but our C_{score} metric has the highest *AUC* value thus being the most favorable metric per the criterion *C*1:*AUC*. While we do not have a quantitative score for *C*2:*Smoothness*, it is evident from Figure 3 that our C_{score} is much closer to a monotonic function than the other metrics whose *TPR* values increase and

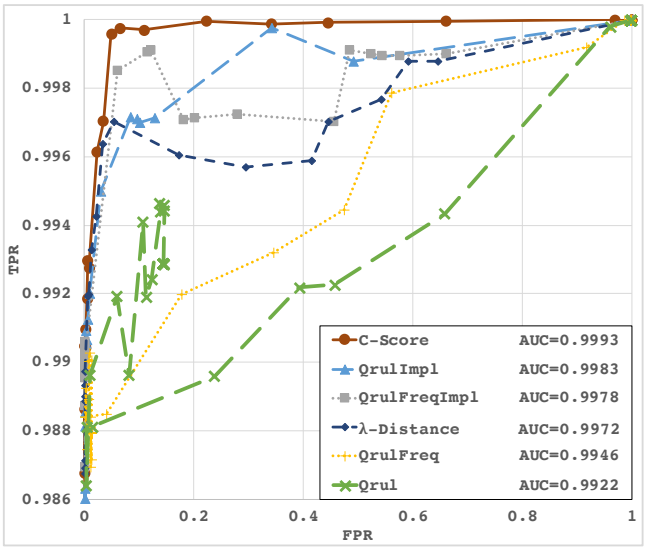

Figure 3: Comparison of Candidate Evaluation Metrics

decrease several times as *FPR* increases. The *Qrul* and *QrulFreq* metrics are particularly poor in terms of smoothness as they both have an inflection point near their weighting variable $\omega`_o = 1$, where increasing the weighting slightly after that point produces instability in the resulting ROC curve. Furthermore, increasing the weighting beyond a certain point causes the metric to only select those candidate rules which have zero over-assignments, resulting in the unterminated portion of the ROC curve for *Qrul* (*QrulFreq* has a similar inflection point that is difficult to discern in Figure 3 at $FPR = 0.0013$).

Unlike the *Qrul* and *λ−Distance* metrics, C_{score} normalizes both the number of logs covered and over-assignments into a ratio between $[0, 1]$ before applying the weighting. This makes the weighting variable independent of the size of the privilege universe and number of log entries and thus easier to understand and apply. In Figure 3, varying the ω weighting of C_{score} between $\omega = \frac{1}{10}$ and $\omega = 10$ varies the charted *FPR* between $FPR = 0.05$ and $FPR = 0.998$ at relatively even intervals. To achieve a similar spread across the *FPR* scores with *QrulFreqImpl* and *λ−Distance*, the weighting variable for those metrics must be varied between $\frac{1}{100}$ and $\frac{1}{2000}$. *QrulImpl* achieved the second highest *AUC* score due to an unusually good score near $FPR = 0.34$, but assigning a weighting to it with predictable results is difficult. For example, the *QrulImpl* score at $FPR = 0.34, TPR = 0.9998$ was achieved with $\omega`_0 = \frac{1}{100000}$, but the next score at $FPR = 0.49, TPR = 0.9988$ was achieved with $\omega`_0 = \frac{1}{500000}$; this significant difference is difficult to determine without the experimentation and consideration of the privilege space and log sizes. Because of its predictability and even distribution of results, C_{score} also best meets our evaluation criterion *C*3:*Interpretability*.

5.2.2 Methods of Calculating CoverageRate. The *CoverageRate* (Formula 3) of the C_{score} (Formula 5) is the number of log entries covered by rule r normalized to the range $[0, 1]$, so that it can be compared with the weighted value of the *OverPrivilegeRate* (Formula 4) normalized to the same range. There are several possible

ways to compute such a coverage rate; however, it is not immediately clear which would perform the best without experimentation. We consider four possible methods for computing *CoverageRate* and analyze their performance here:

- $\frac{|\mathbb{L}_{uncov}(r)|}{|\mathbb{L}_{uncov}|}$: The frequency weighted number of logs covered out of the total number of uncovered logs.
- $\frac{|\{\mathbb{L}_{uncov}(r)\}|}{|\{\mathbb{L}_{uncov}\}|}$: The unique number of logs covered out of the set of unique uncovered logs.
- $\frac{|\mathbb{L}_{uncov}(r)|}{|\mathbb{L}_{OBP}|}$: The frequency weighted number of logs covered out of the total number of logs in the observation period.
- $\frac{|\{\mathbb{L}_{uncov}(r)\}|}{|\{\mathbb{L}_{OBP}\}|}$: The unique number of logs covered out of the set of unique log entries during the observation period.

The results of applying these four methods are presented in Figure 4 with each method identified by its denominator. As evident in Figure 4, the $\frac{|\mathbb{L}_{uncov}(r)|}{|\mathbb{L}_{uncov}|}$ method performed the best for two of our criteria for selecting a candidate metric: *C*1:*AUC* and *C*2:*Smoothness*. The frequency weighted methods $\frac{|\mathbb{L}_{uncov}(r)|}{|\mathbb{L}_{uncov}|}$ and $\frac{|\mathbb{L}_{uncov}(r)|}{|\mathbb{L}_{OBP}|}$ performed about the same in terms of *C*3:*Interpretability* with $\omega = \frac{1}{10}$ resulting in scores in the upper-left most part of the figure. The methods using the number of unique log entries performed less favorably in terms of *C*3:*Interpretability* with their upper-left most points being reached near $\omega = \frac{1}{256}$, a value farther away from 1 and more difficult to find without experimentation. Fluctuations in the trends of $|\{\mathbb{L}_{uncov}\}|, |\mathbb{L}_{OBP}|, |\{\mathbb{L}_{OBP}\}|$ in Figure 4 demonstrate why these methods are poor choices to use for calculating CoverageRate, and are not due to differences in sampling methods or frequency.

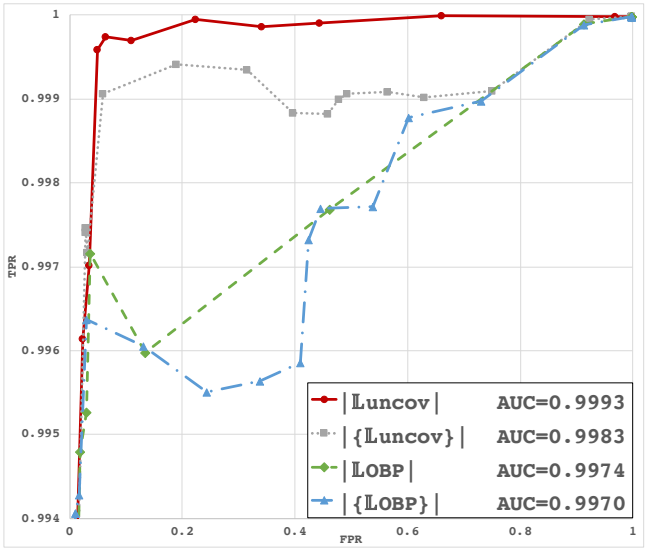

Figure 4: Comparison of CoverageRate Calculating Methods

5.3 Effect of Varying Algorithm Parameters

Besides the ω variable which is varied to generate the points along all the ROC curves (except for the RBAC algorithm curve in Figure 7), two other parameters can be varied as inputs to Algorithm 1: the threshold ϵ used by the *FP−growth* algorithm, and the length of the observation period $|\mathbb{L}_{OBP}|$.

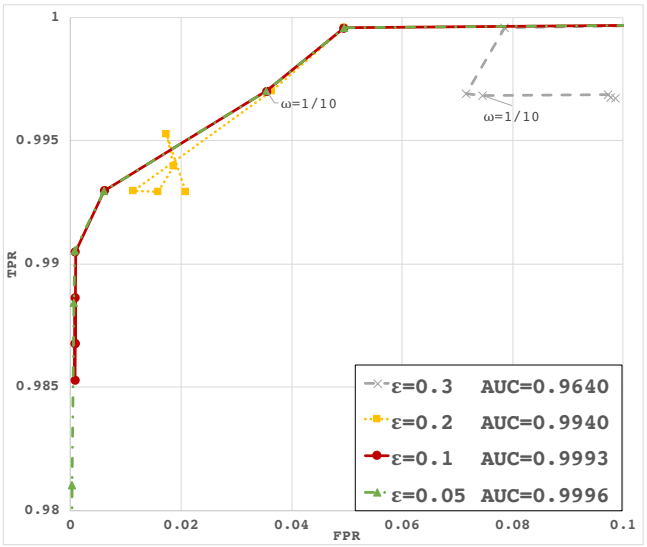

Figure 5: Performance as Itemset Frequency Varies

5.3.1 Effect of Varying Itemset Frequency Threshold. The minimum support threshold (ϵ) is used to specify that a pattern is considered a "frequent" pattern if it occurs in >= ϵ of the examined entries. Increasing ϵ causes fewer candidate patterns to be identified by the *FP−growth* algorithm. The results of varying ϵ between $[0.05, 0.1, 0.2, 0.3]$ are shown in Figure 5. For both $\epsilon = 0.2$ and $\epsilon = 0.3$, we observe inflection points as ω decreases because a lower ω value favors more granular rules in order to lower the over-privilege rate; however, higher ϵ values result in fewer and less granular patterns being identified by the *FP−growth* algorithm. Stated another way, low ω values generally result in lower *FPR* values, while high ϵ values generally result in higher *FPR* values. The inflection points occur as a result of conflicting instructions between low ω and high ϵ values.

Lower ϵ values generate more possible candidates to evaluate and generally result in higher *AUC* scores as well. The trade-off for more candidates however is an increase in run time. At $\omega = \frac{1}{10}$, the average mining times for $\epsilon = 0.05, 0.1, 0.2, 0.3$ were 29.8, 15.3, 2.8, and 1.2 minutes, respectively. Other charts in this section were generated using $\epsilon = 0.1$ as it offered a good trade-off between performance, stability, and run time.

5.3.2 Effect of Varying Observation Period Length. When mining policies with a variable observation period length, a larger observation window generally results in higher *TPR* but also higher *FPR* because intuitively mining algorithms are given more privileges in larger observation periods. This trend is also present with our mining algorithm, albeit not very noticeable. The results of varying the observation period length between $|\mathbb{L}_{OBP}| = [7, 15, 30, 45, 60]$ days are shown in Figure 6. As $|\mathbb{L}_{OBP}|$ increases, TPR generally increases compared to lower $|\mathbb{L}_{OBP}|$ periods of similar *FPR* values,

and the resulting ROC curve becomes smoother. As with ϵ, we observe a trade-off between $|\mathbb{L}_{OBP}|$ and run time. At $\omega = \frac{1}{16}$, the average mining times for $|\mathbb{L}_{OBP}| = [7, 15, 30, 45, 60]$ were 5.7, 6.5, 7.2, 10.5 and 12.8 minutes, respectively. Shorter observation periods such as $|\mathbb{L}_{OBP}| = [7, 15]$ generally produced more fluctuations in the resulting trend lines, which is a common and expected occurrence when using machine learning or data mining techniques with insufficient data. Other charts in this section were generated using $|\mathbb{L}_{OBP}| = 30$ days for a good trade-off between performance, stability, and run time.

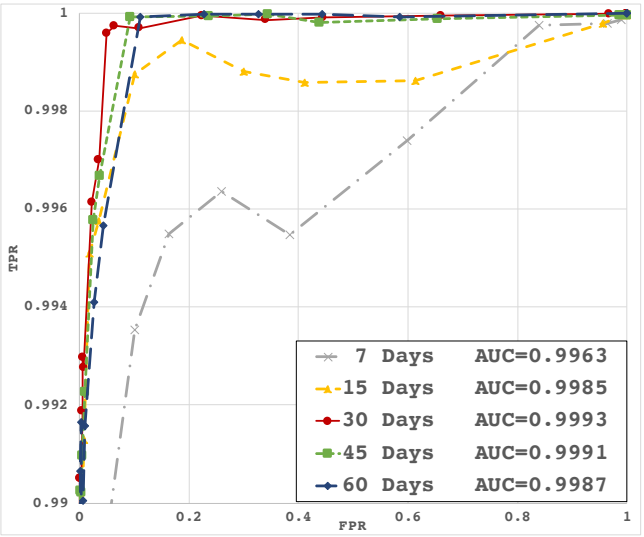

Figure 6: Performance as Observation Period Varies

5.4 ABAC vs. RBAC Performance

We now compare the performance of our ABAC algorithm against an RBAC approach. For this comparison, we use the naive algorithm presented in [16], which builds an RBAC policy based on the permissions exercised during an observation period. Although this RBAC algorithm is fairly simple, it performed quite well in the scenario that sought an equal balance between minimizing under- and over-privilege compared to more sophisticated algorithms [16].

The ROC curve of our ABAC algorithm and the RBAC algorithm from [16] are presented in Figure 7. Our ABAC algorithm used a fixed observation period size of 30 days, an itemset frequency $\epsilon = 0.1$, and the over-privilege weight varied between $\omega = [\frac{1}{8192}, ..., 16]$ by powers of 2 to generate the data points. For the RBAC algorithm, there is no variable similar to ω that can be used to instruct the algorithm to directly vary the importance between under- and over-privilege. However, varying the observation period length effectively serves this purpose by causing more or fewer privileges to be granted by the algorithm, so the observation period length was varied between [3, 7, 15, 30, 45, 60, 75, 90, 105, 120] days to generate the data points for the RBAC algorithm in Figure 7.

Our ABAC algorithm significantly outperformed the RBAC algorithm across the ROC curves in Figure 7. With only 30 days worth of data, the ABAC algorithm was able to correctly grant

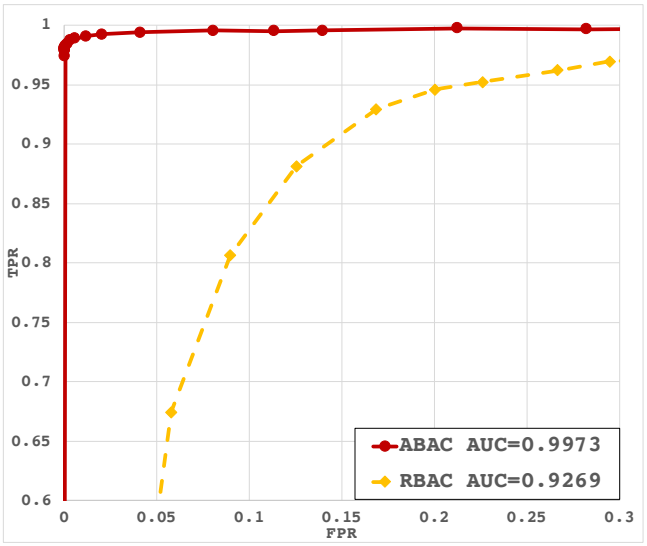

Figure 7: Comparison of ABAC vs. RBAC Performance

more privileges (higher TPR) than the RBAC algorithm with 120 days of data. The ABAC algorithm was also able to correctly restrict more unnecessary privileges (lower FPR) than the RBAC algorithm operating on only 3 days of data. This is due to the ability of the ABAC algorithm to identify patterns and create policies based on attributes vs. the RBAC algorithm which is restricted to using only RBAC semantics.

6 CONCLUSION

This paper explored an approach for automatically generating least privilege ABAC policies that balance between minimizing under- and over-privilege assignment errors. We formally defined the ABAC Privilege Error Minimization Problem ($ABAC_{PEMP}$). We took an unsupervised rule mining approach to design an algorithm which automatically performs ABAC policy generation by mining audit logs with a variable weighting between under- and over-privilege. We designed a policy scoring algorithm for evaluating ABAC policies from the least privilege perspective by using out-of-sample validation. We designed performance optimization methods including feature selection, partitioning, and parallelization to address the challenges of large ABAC privilege spaces. Finally, we presented the results of applying our approach on a real-world dataset to demonstrate its effectiveness and its better performance than an RBAC approach. The algorithms and methods that we developed in this work do not depend on the system and the organization from which we obtained the valuable dataset. They could be adopted by any organization to start their ABAC policy generation and deployment as we highlighted at the beginning of Section 4. Overall, we hope our work can help promote a wider and faster deployment of the ABAC model, and can help unleash the advantages of ABAC to better protect large and complex computing systems.

ACKNOWLEDGMENTS

This research was supported in part by the NSF grant OIA-1936968.

REFERENCES

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, Vol. 1215. 487–499.

[2] Amazon Web Services. 2019. AWS CloudTrail. https://aws.amazon.com/cloudtrail/. Accessed: 2019-06-09.

[3] Amazon Web Services. 2019. AWS CloudTrail Log File Examples. https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-log-file-examples.html. Accessed: 2019-06-09.

[4] Amazon Web Services. 2019. AWS Products and Services. https://aws.amazon.com/products/. Accessed: 2019-06-09.

[5] Lujo Bauer, Scott Garriss, and Michael K Reiter. 2011. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (2011), 2.

[6] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery* 8, 1 (2004), 53–87.

[7] Trevor Hastie, Jerome Friedman, and Robert Tibshirani. 2001. *The elements of statistical learning*. Springer series in statistics New York, NY, USA.

[8] Vincent C Hu et al. 2013. NIST 800-162: Guide to attribute based access control (ABAC) definition and considerations (Draft).

[9] Carlos Cotrini Jiménez, Thilo Weghorn, and David A. Basin. 2018. Mining ABAC Rules from Sparse Logs. *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), 31–46.

[10] John D. Kelleher, Brian Mac Namee, and Aoife D'Arcy. 2015. *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. MIT Press.

[11] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. 2008. Mining roles with semantic meanings. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*.

[12] Ian Molloy, Youngja Park, and Suresh Chari. 2012. Generative Models for Access Control Policies: Applications to Role Mining over Logs with Attribution. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*.

[13] Linux Kernel Organization. 2019. Ext4 Disk Layout. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html. Accessed: 2019-09-14.

[14] Carlos E. Rubio-Medrano, Josephine Lamp, Adam Doupé, Ziming Zhao, and Gail-Joon Ahn. 2017. Mutated Policies: Towards Proactive Attribute-based Defenses for Access Control. In *Proceedings of the Workshop on Moving Target Defense*.

[15] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[16] Matthew W Sanders and Chuan Yue. 2018. Minimizing Privilege Assignment Errors in Cloud Services. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*. 2–12.

[17] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47.

[18] SANS Institute. 2019. A Compliance Primer for IT Professionals. https://www.sans.org/reading-room/whitepapers/compliance/compliance-primer-professionals-33538. Accessed: 2019-06-09.

[19] Harold F Tipton and Kevin Henry. 2006. *Official (ISC) 2 guide to the CISSP CBK*. Auerbach Publications.

[20] Zhongyuan Xu and Scott D Stoller. 2014. Mining attribute-based access control policies from logs. In *Proceedings of the IFIP DBSec*. Springer, 276–291.

[21] Zhongyuan Xu and Scott D Stoller. 2015. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing* 12, 5 (2015).

A APPENDIX ON AUDIT LOG EXAMPLES

A.1 Amazon EC2 Log Entry Example

Figure 8 is an Amazon EC2 log example [3]. It shows user Alice started an EC2 virtual machine instance ("eventName": "StartInstances") with the instanceId "i-ebeaf9e2". No "errorCode" field is present in the log entry, so we can tell that the request succeeded. From the response elements we can tell that the virtual machine instance has been moved from the "stopped" state into the "pending" state which indicates it is starting up. The log provides several other attributes such as eventTime, awsRegion, sourceIPAddress, and userAgent that may be used by an ABAC algorithm. In this log entry we see "awsRegion":"us-east-2". If, for example, Alice consistently creates instances in "us-east-2" only, a mining algorithm can use this information to create a policy which only allows Alice to create instances in that region.


```
{"Records": [{
    "eventVersion": "1.0",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::123456789012:user/Alice",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "accountId": "123456789012",
        "userName": "Alice"
    },
    "eventTime": "2014-03-06T21:22:54Z",
    "eventSource": "ec2.amazonaws.com",
    "eventName": "StartInstances",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "205.251.233.176",
    "userAgent": "ec2-api-tools 1.6.12.2",
    "requestParameters": {"instancesSet":
            {"items": [{"instanceId": "i-ebeaf9e2"}]}},
    "responseElements": {"instancesSet": {"items": [{
        "instanceId": "i-ebeaf9e2",
        "currentState": {
            "code": 0,
            "name": "pending"
        },
        "previousState": {
            "code": 80,
            "name": "stopped"
        }
    }]}}
}]}
```


Figure 8: AWS EC2 Log Entry Example

A.2 AWS IAM Log Entry Example

Figure 9 is an Amazon IAM log example [3]. It shows user Alice created a user ("eventName": "CreateUser") with the username "Bob". Again, no "errorCode" field is present in the log entry, so we can tell that the request succeeded. From the userAgent field in this log entry we see that Alice is using the aws-cli (i.e., AWS Command Line Interface) to perform this operation. Some operations are more likely to be run from the CLI or in code from automated tools. Such information can help an ABAC policy miner create policies which grant requests based on the user agent being used.


```
{"Records": [{
    "eventVersion": "1.0",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::123456789012:user/Alice",
        "accountId": "123456789012",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
    },
    "eventTime": "2014-03-24T21:11:59Z",
    "eventSource": "iam.amazonaws.com",
    "eventName": "CreateUser",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "127.0.0.1",
    "userAgent": "aws-cli/1.3.2 Python/2.7.5 Windows/7",
    "requestParameters": {"userName": "Bob"},
    "responseElements": {"user": {
        "createDate": "Mar 24, 2014 9:11:59 PM",
        "userName": "Bob",
        "arn": "arn:aws:iam::123456789012:user/Bob",
        "path": "/",
        "userId": "EXAMPLEUSERID"
    }}
}]}
```


Figure 9: AWS IAM Log Entry Example

B APPENDIX ON ABAC POLICY EXAMPLE

Figure 10 is an example ABAC rule generated by our Algorithm 1 and written in JSON format. This rule allows three specific users (actual user names redacted) to run any operation of the AWS CloudFormation service when those users are MFA authenticated (i.e., they passed multi-factor authentication). AWS CloudFormation is typically used only by administrators for describing and provisioning infrastructure resources in a cloud environment. The facts that the access key is none, the userAgent is the service name, and the eventType is "AwsApiCall" all restrict these operations to be run from the AWS web console and not from any code or CLI (Command Line Interface).

```
{
  "sourceIPAddress": ["cloudformation.amazonaws.com"],
  "userIdentity.sessionContext.attributes.mfaAuthenticated": ["true"],
  "userIdentity.accessKeyId": ["NONE"],
  "userIdentity.userName": ["USER1","USER2","USER3"],
  "userAgent ": ["cloudformation.amazonaws.com"],
  "eventType": ["AwsApiCall"]
}
```

Figure 10: Example ABAC Rule Generated by Algorithm 1

This generated ABAC rule can be deployed to systems that support ABAC policy deployment and enforcement. AWS has some ABAC support by using "condition" elements. For example, this ABAC rule can be deployed as an AWS IAM (Identity and Access Management) policy as shown in Figure 11.

```
{
  "Version": "2017-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
         "cloudformation:*",
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {"aws:username": ["USER1","USER2","USER3"]},
        "BoolIfExists": {"aws:MultiFactorAuthPresent": true}
      }
    }
  ]
}
```

Figure 11: IAM Policy Deployment of the Rule in Figure 10