Mining ABAC Rules from Sparse Logs

Carlos Cotrini
Department of Computer Science
ETH Zurich, Switzerland
Email: ccarlos@inf.ethz.ch

Thilo Weghorn
Department of Computer Science
ETH Zurich, Switzerland
Email: tweghorn@inf.ethz.ch

David Basin
Department of Computer Science
ETH Zurich, Switzerland
Email: basin@inf.ethz.ch

***Abstract*—Different methods have been proposed to mine attribute-based access control (ABAC) rules from logs. In practice, these logs are sparse in that they contain only a fraction of all possible requests. However, for sparse logs, existing methods mine and validate overly permissive rules, which allow escalation of privileges. We define a novel measure, *reliability*, that quantifies how overly permissive a rule is and we show why other standard measures like confidence and entropy fail in quantifying over-permissiveness. We build upon state-of-the-art subgroup discovery algorithms and our new reliability measure to design Rhapsody, the first ABAC mining algorithm with correctness guarantees: Rhapsody mines a rule if and only if the rule covers a significant number of requests, its reliability is above a given threshold, and there is no equivalent shorter rule. We evaluate Rhapsody on different real-world scenarios using logs from Amazon and a computer lab in a large university. Our results show that Rhapsody generalizes better and produces substantially smaller rules than competing approaches.**

1. Introduction

Attribute-based access control (ABAC) is an upcoming access control standard. The National Institute of Standards and Technology (NIST) issued in 2014 a special publication recommending ABAC over role-based access control (RBAC) and access control lists [1]. Gartner Inc. estimates that by 2020 70% of all businesses will use ABAC to protect critical assets [2], [3].

1.1. Problem context

The specification and maintenance of ABAC rules brings new challenges. Manually migrating to ABAC is more difficult than migrating to RBAC, which is already considered to be time-consuming and cumbersome [1]. Even after specification, an ABAC rule set must be maintained and audited, as organizational changes like mergers and acquisitions can make the rules convoluted or inaccurate.

An alternative to manually specifying or maintaining an ABAC rule set is to automatically *mine* ABAC rules from access logs [4]–[7]. Since logs reflect both the implemented access control rules and user behavior within the organization, mining from logs can help to refactor and simplify rules that become overly-complex due to organizational changes. It can also help to identify *overly permissive rules*; that is, rules that assign permissions to users who, according to the log, are not using them.

In this paper, we examine one aspect of this problem that is essential to using ABAC rule mining in practice: *mining ABAC rules from sparse logs*. Logs reflect expected requests as users generally access resources they are granted. In consequence, real world logs typically contain only a small subset of all possible access requests. We consider three logs in our case studies and all of them contain less than 10% of all possible requests.

This problem setting subsumes other previously considered settings, like mining from non-sparse logs and mining where permissions are given by formalisms such as access control matrices.

1.2. Limitations of the state-of-the-art

Despite intensive research in this area [8]–[15], previous work has serious limitations.

Sparse logs. When the logs are sparse, some competing approaches [4], [5] cannot mine useful ABAC rules. This is because these approaches are intended only for mining from access control matrices or from logs containing a large fraction of all possible access requests.

Over-permissiveness. Mining algorithms [14], [15] resort to quality measures like confidence [16] and weighted relative accuracy [17] to guide the search and selection of rules. We show in Section 5.2 that, for sparse logs, these measures lead to the mining of *overly permissive* rules. These are rules that cover a significant number of requests without any evidence of authorization in the log, allowing escalation of privileges.

Rule size. Current algorithms produce unnecessarily large rules. Succinct rules are desirable from the administrative perspective as they are easier to audit and maintain.

Cross-validation. When evaluating models learned from logs, the standard cross-validation method splits the log into a training log and a testing log. We show that this approach validates overly permissive rules, which is undesirable from the security perspective. This happens because the logs reflect expected access requests. Denied access requests are therefore rare and mostly due to human error, rather than malice. A testing log does not include a representative set of requests that would be issued by malicious users. As a result, evaluating a mined rule set on such a testing log validates overly permissive rules.

1.3. Our approach

Rhapsody. We introduce a new ABAC mining algorithm that address the first three limitations of the state-of-the-art as follows.

Sparse logs. Rhapsody uses ideas from association rule mining [16], [18] to mine from sparse logs.

No overly permissive rules. Rhapsody avoids mining overly-permissing rules by using *reliability* (see Definition 7), a new rule quality measure, to guide the mining of rules. We demonstrate that reliability gives low scores to rules that are overly permissive or have low confidence (see Observation 1 in Section 3.2). We also show that standard rule quality measures cannot quantify over-permissiveness.

Rule size. Rhapsody only mines rules that have no shorter equivalent rule.

Moreover, Rhapsody guarantees to mine exactly all rules that have a reliability above a given threshold and have no shorter equivalent rule (see Theorem 1 in Section 4.2). Table 1 shows how Rhapsody improves upon state-of-the-art mining algorithms.

Algorithm	Sparse logs	No overly perm. rules	Shortest rules
Xu & Stoller's miner [8]	✗	✓	✗
APRIORI-C [14]	✓	✗	✓
APRIORI-SD [15]	✓	✗	✓
Classsification-tree [19]	✓	✓	✗
CN2 [12]	✓	✓	✗
Rhapsody	✓	✓	✓

TABLE 1: State-of-the-art ABAC policy mining algorithms.

Universal cross-validation. We present an alternative to cross-validation for evaluating ABAC mining algorithms. In contrast to standard cross-validation, universal cross-validation evaluates mined policies with requests not occurring in the log. In this way, universal cross-validation improves the quality of the policies mined by *any* ABAC mining algorithm, as it identifies those policies containing overly permissive rules.

1.4. Case studies and evaluation

We experimentally compare Rhapsody with other methods for mining ABAC policies from logs. Our experimental evaluation is the most comprehensive to date in access control mining. We use logs from a large university and two logs provided by Amazon [20], [21], where the last two are available for research purposes. Previous researchers used only logs from one enterprise, which are not publicly available [9], [22] or just synthetic data [5], [7].

Our experimental results, presented in Section 6.4 and summarized in Table 1, illustrate the four limitations of the state-of-the-art and demonstrate that Rhapsody and universal cross-validation overcome them.

Sparse logs. Rhapsody mines policies of higher quality than Xu and Stoller's approach [4], [5], [8]. When mining from *sparse logs*, Xu and Stoller's mined policies have true positive rates close to 0%. In contrast, Rhapsody's policies attain true positive rates above 80% in most of the cases.

Over-permissiveness. Subgroup-discovery algorithms like APRIORI-SD and APRIORI-C mine *overly-permissive rules,* which yield false positive rates above 10%. By changing these algorithms' input parameters, one can decrease the false positive rate to values close to 0% but at the cost of true positive rates below 40%. In contrast, Rhapsody's mined policies attain false positive rates close to 0% and true positive rates above 80% in most of the cases.

Rule size. Classification-tree learners [10], [19] and CN2 [12], [23] mine *unnecessarily large rules.* For the Amazon logs, the rules mined by Rhapsody are at least 40% shorter and in some cases they are even 90% shorter.

Cross-validation. In 80% of the cases, universal cross-validation validates policies with F1 scores (a standard generalization metric [24]) greater or equal than those from policies validated by standard cross-validation. In most of the cases, the F1 scores improve in at least 30%. This improvement holds for *any mining algorithm*.

1.5. Contributions

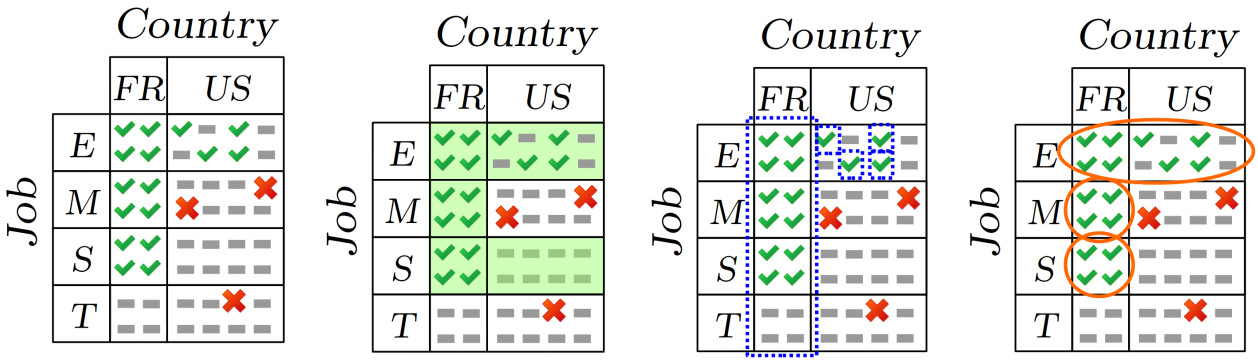
In summary, we make the following contributions.

Mining algorithm for sparse logs. We propose *Rhapsody*, a new ABAC mining algorithm that, in comparison with previous work, mines policies from sparse logs that generalize better.

Quality measure for over-permissiveness. We introduce *reliability*, a new measure that quantifies how overly permissive a rule is. We show that reliability only gives high values to rules that are not overly permissive and show why other standard measures fail to measure over-permissiveness.

Succinctness and correctness. Rhapsody *guarantees* that if there is a succinct rule satisfied by a significant number of requests and with a reliability above a given threshold, then this rule will be mined (See Theorem 1).

Evaluation methodology. We propose *universal cross-validation*, a new validation method that, in comparison with cross-validation on logs, validates policies with substantially higher F1 scores.



(a) An organization with an access log.

(b) Security policy.

(c) Implemented policy.

(d) Mined policy.

Figure 1: An illustrative example for ABAC mining. Each ✓, ✗, and ▬ denotes a request (i.e., user). The ticks ✓ and crosses ✗ denote logged requests that have been authorized and denied, respectively. The tiny rectangles ▬ denote users who have not requested the permission yet.

The remainder of this paper is organized as follows. In Section 2 we provide background on the ABAC mining problem. In Section 3 we show how standard rule quality measures give high scores to overly permissive rules and we introduce a better measure, reliability. In Section 4 we use our reliability measure to build Rhapsody, a new ABAC mining algorithm. In Section 5 we show how cross-validation on logs validates overly permissive rules and present universal cross-validation. In Section 6 we experimentally compare Rhapsody with other ABAC mining algorithms and universal cross-validation with cross-validation. Finally, in Sections 7 and 8 we discuss related work and draw conclusions.

2. The ABAC mining problem

We begin by describing the setting and the objectives for ABAC mining. Afterwards, we formalize ABAC's syntax and semantics and the ABAC mining problem.

2.1. Setting

An ABAC *policy* is a set of *rules*, where each rule describes a set of conditions based on the *attribute values* of *users* and *permissions* [25], [26]. A permission denotes an action on an object, and so a permission's attributes refer to the attributes of both the object and the action. Whenever a user wants to exercise a permission, she must issue a *request* identifying herself and the permission. The request contains the user's and the permission's attribute values. A request is *authorized* by an ABAC policy iff the request satisfies all the conditions of at least one of the policy's rules. We assume the existence of a *log* that records every issued request and whether the request was authorized or not.

We describe a setting that motivates ABAC mining. Organizations define *(organizational) security policies* expressing which requests should be authorized. For enforcement purposes, the policy administrator must specify this as an ABAC policy in a machine-readable way for the organization's IT systems. We call this the *implemented ABAC policy*.

Due to organizational changes or human errors made during the policy's implementation, mistakes may be introduced in the implemented policy. These can be classified into two types: incorrect authorizations (i.e., requests authorized by the implemented policy but not by the security policy) or incorrect denials (i.e., requests authorized by the security policy but not by the implemented policy).

Incorrect authorizations are the hardest to detect and the most problematic because they may be used for nefarious purposes, e.g., to read confidential data, escalate privileges, and in general to attack systems. In contrast, incorrect denials are usually not so problematic from the security perspective; when users discover that valid requests are not authorized, they report them to the policy administrator, who must then manually add exceptions to the implemented policy. However, adding these exceptions is time-consuming and their addition also makes the implemented policy convoluted.

2.2. Illustrative example

Figure 1 illustrates our setting. Figure 1a describes an organization with 48 users, one permission, and an access log. Each ✓, ✗, and ▬ denotes a request. Since there is only one permission, we can identify each request with the user who issues it. The ticks ✓ and crosses ✗ denote logged requests (i.e., users) that have been authorized and denied, respectively. The tiny rectangles ▬ denote users who have not requested the permission yet. Users have three attributes: $Country$, Job, and ID. $Country$ can take the values US and FR whereas Job can take the values E, M, S, and T (they stand for Engineer, Manager, Secretary, and Technician, respectively). The ID of each user and the permission are not shown in the figures.

Figure 1b describes the same organization from Figure 1a but with an organizational security policy,

described by the shaded rectangles. This security policy encloses all requests that should be authorized. According to this policy, all engineers, secretaries, and French managers should be authorized.

Figure 1c describes the actual implemented ABAC policy, represented with dotted rectangles, which authorizes all French users and four US-based engineers. Observe that this policy incorrectly authorizes all French technicians and incorrectly denies requests from US-based secretaries and some US-based engineers. The four authorized US-based engineers represent users who were initially denied authorization, reported these incorrect denials to the administrator, and were later added as exceptions to the implemented policy.

Our goal is to propose an *ABAC mining algorithm* that, given a log of requests, mines ABAC rules that identify patterns among the authorized requests. Our ABAC mining algorithm, Rhapsody, would mine the rules given by the ovals in Figure 1d.

2.3. Use cases for ABAC mining

ABAC mining has the following applications:

Identification of missing rules. The mined rules can find patterns among the manually added exceptions to discover missing rules in the currently implemented policy, like the rule authorizing US-based engineers in Figure 1b.

Identification of overly permissive rules. The mined rules can be compared with the currently implemented policy to identify overly permissive rules that authorize requests for which the log gives insufficient evidence. In our example, the rule authorizing all French users in Figure 1c is overly permissive; it authorizes all French technicians, but none of them have requested access so far. The policy administrator can discover this by comparing the mined policy in Figure 1d with the implemented policy and then he can decide if French technicians really need to be authorized.

Refactorization. The mined rules can be used to refactor an old policy that has become convoluted after organizational changes.

Migration to ABAC. An ABAC mining algorithm can also be used to mine ABAC policies from other configurations like access control lists or RBAC models.

2.4. Objectives of ABAC mining

Based on the previous discussion, we now state the properties a mining algorithm should satisfy to be useful.

Generalization. The mining algorithm can simply *overfit* the log and mine a policy that authorizes precisely the authorized requests in the log and nothing else. This policy is not useful as it does not help to explain what is missing in the implemented policy. For this reason, a mining algorithm should return policies that *generalize well*; that is, the policies also authorize non-logged requests for which a significant number of similar requests have been authorized. A standard method for evaluating how well a mined policy generalizes is cross-validation [27], which we discuss in Section 5.

Precision. The algorithm should not mine policies authorizing sets of requests for which the log offers no evidence, like the rule authorizing all French technicians in Figure 1c. For the case of French technicians, the algorithm should conservatively risk an incorrect denial rather than an incorrect authorization, as the latter is harder to detect and more security critical. The absence of these requests in the log shows that these requests are infrequent, so the work required by the policy administrator in case of an incorrect denial remains low.

Succinctness. The mined rules are used by the policy administrator to correct the implemented policy. Therefore, the mined rules should be succinct, so that the policy administrator can easily understand them. For example, if a rule states that "all US-based technicians are authorized" and we know that all technicians are from the US, then a more preferable rule would be "all technicians are authorized."

2.5. ABAC syntax and semantics

ABAC policies are positive formulas expressing (binary) relationships between users and permissions. They can generally be defined in a restricted fragment of sorted first-order logic. This is sufficient to cover all ABAC scenarios encountered in the literature.

Definition 1. (ABAC syntax) We fix a set of sorts and a signature Σ of finitely many sorted constant, function, and predicate symbols. We assume that the set of sorts includes the two distinguished sorts $Users$ and $Permissions$, as well as other sorts typically used in ABAC like integers, strings, or sets of strings.

We define the set of ABAC policies using the following grammar, which defines a subset of quantifier-free, positive, first-order formulas:

$$
\begin{array}{lll}
t & ::= f(\mathbf{u}) \mid f(\mathbf{p}) \mid c & \text{/* Terms */} \\
\alpha & ::= t = t \mid Q(t,t) & \text{/* Atoms */} \\
r & ::= \alpha \mid \alpha \wedge r & \text{/* Rules */} \\
\pi & ::= \{r, \ldots, r\}. & \text{/* Policies */}
\end{array}
$$

Here, $\mathbf{u}$ and $\mathbf{p}$ range over variables of sorts $Users$ and $Permissions$, respectively, whereas f, c, and Q range over unary function, constant, and binary relation symbols of the appropriate sorts, respectively. We restrict atoms, rules, and policies to have at most two free variables of sorts $Users$ and $Permissions$, respectively.

An expression of the form α, r, or π is called an (ABAC) *atom*, *rule*, or *policy*, respectively. Any unary function symbol f with domain $Users$ or $Permissions$ is an *attribute*. If the domain is $Users$, we call f a *user attribute*; otherwise, f is a *permission attribute*. The *size* of an ABAC policy or a rule is the number of atoms occurring in it.

Definition 2. (ABAC semantics) As is standard for a sorted first-order language, ABAC's semantics is given by three components. First, a collection of carrier sets, one for each sort. This includes, in particular, a set U of users and a set P of permissions. Second, a function $\mathfrak{I}$ that interprets every symbol in the signature, e.g., each attribute f corresponds to a unary function $f^{\mathfrak{I}}$ over the appropriate carrier sets. Finally, a sort-respecting substitution σ mapping the variables $\mathbf{u}$ and $\mathbf{p}$ to elements in U and P, respectively. We denote variables with bold letters and elements of carrier sets with italicized letters.

We call a pair in $U \times P$ a *request*. For a user attribute f and $u \in U$, the value $f^{\mathfrak{I}}(u)$ is called *u's attribute value for f*. For a permission attribute f and $p \in P$, the value $f^{\mathfrak{I}}(p)$ is called *p's attribute value for f*. For an atom α, we denote with $\mathfrak{I}, \sigma \vDash \alpha$ the standard first-order satisfiability relation. A rule r *authorizes* or *covers* a request (u, p) if $\mathfrak{I}, \sigma \vDash \alpha$, for every atom α occurring in r. A policy π *authorizes* a request (u, p) if some rule in π authorizes it. For a rule r and a set $S \subseteq U \times P$, we let $[\![r]\!]_S$ be the set of requests in S authorized by r. We now fix U, P, $\mathfrak{I}$, and σ for the rest of the paper.

2.6. ABAC mining

We now formally define the ABAC mining problem. We start by defining an ABAC instance as a structure describing the set of users, the set of permissions, and the set of requests that have been logged so far in an access control system.

Definition 3. (ABAC instance) An *ABAC instance* is a tuple (U, P, A, D) where U and P are sets representing all users and permissions in an organization, A and D are disjoint subsets of $U \times P$ and they denote the set of authorized and denied requests, respectively. The *log* of the instance is the set $A \cup D$.

In *the ABAC mining problem*, we are given as input an ABAC instance (U, P, A, D) and the objective is to find a *precise* ABAC policy of *minimal size* that *generalizes well*. The policy's size is measured as described in Definition 1. To measure the precision and how well the policy generalizes, we use *universal cross-validation*, a new approach to cross-validation, introduced in Section 5. Algorithms intended to solve this problem are called *ABAC mining algorithms*.

3. Quantifying over-permissiveness

Most of ABAC mining algorithms work by computing a set of candidate rules and selecting those that have a high score according to some rule quality measure. State-of-the-art quality measures depend only on the *confidence* of the rule (the ratio of authorized requests covered by the rule to the total requests covered by the rule). As we shall see, this is insufficient and these measures may give high scores to rules that we denote as *overly permissive*: rules that authorize significant sets of requests with insufficient evidence from the log. Since these rules are undesirable, we propose a new quality measure for rule selection: *reliability*. We prove that reliability gives a high value to a rule iff it has a high confidence and, in addition, is *not* overly permissive (see Observation 1).

As a motivating example, consider the ABAC instance depicted in Figure 1a. From our discussion in Section 2.4, an ABAC mining algorithm should mine the rule $Job(\mathbf{u}) = E$, but not the rule $Country(\mathbf{u}) = FR$. The log shows that at least half of the engineers have requested access and been authorized. Since users with the same attribute values generally perform the same functions in an organization, they should also have the same permissions. Therefore, $Job(\mathbf{u}) = E$ should be mined. In contrast, regarding the rule $Country(\mathbf{u}) = FR$, although most of the French users have been authorized, those who have not requested authorization yet are precisely those who are technicians. There is no evidence in the log yet to conclude that all French technicians should be authorized. Therefore, $Country(\mathbf{u}) = FR$, should not be mined at this point.

Surprisingly, as we show next, current measures give a higher value to $Country(\mathbf{u}) = FR$ than to $Job(\mathbf{u}) = E$. Moreover, there are mining algorithms like APRIORI-SD [15] and APRIORI-C [14] that, when given as input the ABAC instance of Figure 1a, mine the rule $Country(\mathbf{u}) = FR$. They mine this rule even when all French technicians are marked as denied in the log.

3.1. Over-permissiveness

We start with some definitions. Recall that, for an ABAC instance (U, P, A, D), a subset $S \subseteq U \times P$, and a rule r, the set $[\![r]\!]_S$ consists of all requests in S satisfying r.

Definition 4 (Confidence [16])**.** Let (U, P, A, D) be an ABAC instance. The *confidence* of a rule r is

$$Conf(r) := \frac{|[\![r]\!]_A|}{|[\![r]\!]_{U \times P}|}.$$

Previous mining algorithms used confidence to measure a rule's quality [14]–[16], [18]. If a rule's confidence is high, then a large fraction of the requests covered by the rule has been authorized. According to these algorithms, high confidence indicates that all the other covered requests should also be authorized.

In Figure 1a, the rules $Country(\mathbf{u}) = FR$ and $Job(\mathbf{u}) = E$ have confidence 0.75 and 0.66, respectively.

Definition 5. For a rule r, we call a *refinement of* r any rule of the form $r \wedge r'$, for some rule r'.

A rule's refinement identifies subsets covered by the rule. Since we assume only signatures with finitely many symbols, a rule has only finitely many refinements.

Two refinements of the rule $Country(\mathbf{u}) = FR$ are $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = E$ and $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = T$. The confidence of these refinements are 1.0 and 0.0, respectively.

Although $Country(\mathbf{u}) = FR$ has a high confidence, a good ABAC mining algorithm should not mine this

rule; it authorizes all French technicians, but none of them has even requested the permission. More precisely, $Country(\mathbf{u}) = FR$ has the refinement $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = T$ covering a significant set of requests, but with confidence 0. To describe these kind of rules, we introduce the following concept.

A rule is overly permissive *if one of its refinements covers a significant set of requests but has low confidence.*

An overly permissive rule goes against the principle of least privilege and should be replaced with rules that avoid the low-confidence refinement. In the case of Figure 1a, a policy containing the rule $Country(\mathbf{u}) = FR$ should have instead the rules $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = E$, $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = M$, and $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = S$.

To formally define over-permissiveness, we must agree on when a set of requests is significant and when a refinement has low confidence. These notions are not absolute and they depend on the ABAC instance. Hence, we let the policy administrator specify parameters T and K, which define when a set of requests is significant enough and when a refinement has low confidence.

Definition 6. Let $T \geq 1$ and $K \in [0, 1]$. A rule is *overly permissive with respect to T and K* if there is a refinement $r \wedge r'$ of r with $|[\![r \wedge r']\!]_{U \times P}| \geq T$ and $Conf(r \wedge r') < K$.

The values for T and K must be given as input to Rhapsody, so that Rhapsody can decide when a rule is overly permissive. We omit them when they are clear from the context.

In our experiments in Section 6 we found that, for an ABAC instance (U, P, A, D), a good value for K is around $|A| / |U \times P|$. Very high values are too harsh and many promising rules would be regarded as overly permissive. In a sparse log, there are many requests that have not been evaluated yet, so a refinement rarely has very high confidence. Analogously, very low values are too lenient and refinements with low confidence would not be regarded as overly permissive.

Regarding good values for T, one could argue that, in general, the best is 1, because this ensures that all refinements are considered. However, we are interested only in refinements that cover a significant number of requests. In ABAC instances with thousands of users and permissions and sparse logs, one can easily find refinements that cover 1 or 2 requests with confidence 0. Setting $T = 1$ could, therefore, unfairly mark promising rules as overly permissive. In general, the lower T is, the more likely it is for a rule to be overly permissive, as more refinements are considered.

We now discuss suitable values for T and K for Figure 1a. The smallest refinement here has size 4, so we can let $T = 4$ to ensure that all significant refinements are considered when evaluating if a rule is overly permissive. For K, we can use $\frac{|A|}{|U \times P|} \approx 0.3$. If a refinement has a confidence below 0.3, then we cannot be convinced that all requests in that refinement should be authorized. So, for this ABAC instance, we fix then $T = 4$ and $K = 0.3$.

With the above choice of T and K, the rule $Country(\mathbf{u}) = FR$ is overly permissive because one of its refinements, namely $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = T$, covers 4 requests but has confidence 0. In contrast, the rule $Job(\mathbf{u}) = E$ is not overly permissive, since its two refinements have confidence above 0.3.

3.2. Reliability

Table 2 illustrates rule quality measures from state-of-the-art mining algorithms. We can see that all of them depend just on the confidence of the rule, the total requests, and the total authorized requests. One can easily verify that the value of these measures is high whenever the confidence is high.

Quality measure		Formula
Support	[16]	$Supp(r) = \lvert[\![r]\!]_{U \times P}\rvert$
Confidence	[16]	$Conf(r) = \frac{\lvert[\![r]\!]_A\rvert}{\lvert[\![r]\!]_{U \times P}\rvert}$
Likelihood ratio statistic	[12]	$2Supp(r)\ Conf(r) \log\left(\frac{Conf(r)}{\left(\frac{\lvert A\rvert}{\lvert U \times P\rvert}\right)}\right) +$ $2Supp(r)\,(1 - Conf(r)) \log\left(\frac{1-Conf(r)}{\left(1-\frac{\lvert A\rvert}{\lvert U \times P\rvert}\right)}\right)$
Entropy	[28]	$Conf(r) \log(Conf(r)) +$ $(1 - Conf(r)) \log(1 - Conf(r))$
WRAcc	[17]	$\frac{Supp(r)}{\lvert U \times P\rvert}\left(Conf(r) - \frac{\lvert A\rvert}{\lvert U \times P\rvert}\right)$
Gini index	[10]	$-Conf(r)\,(1 - Conf(r))$

TABLE 2: Quality measures proposed for rule mining, which are all based on $Conf(r)$, $Supp(r)$, $|A|$, and $|U \times P|$.

Despite extensive research on these measures, *none of them is able to quantify both confidence and over-permissiveness*. For example, in Figure 1a, all measures give a value to $Country(\mathbf{u}) = FR$ that is higher than the value given to $Job(\mathbf{u}) = E$. However, as we discussed above, $Country(\mathbf{u}) = FR$ is overly permissive and $Job(\mathbf{u}) = E$ is not. More generally, for any measure and any choice of T and K, there are scenarios where the measure gives high values to overly permissive rules.

Motivated by these limitations, we propose *reliability*, a measure that quantifies not only the confidence of a rule, but also the confidence of all its significant refinements.

Definition 7. (Reliability) Let (U, P, A, D) be an ABAC instance. For $T \geq 1$, the *T-reliability of a rule r* is:

$$Rel_T(r) := \min_{r' \in F_T(r)} Conf(r \wedge r')\,.$$

where $F_T(r) = \{r' : |[\![r \wedge r']\!]_{U \times P}| \geq T\}$. In the degenerate case of $F_T(r) = \emptyset$, define $Rel_T(r) := Conf(r)$.

The parameter T corresponds to the same parameter T in the definition of over-permissiveness. The following

observation, which has a straightforward proof, gives the connection between reliability and over-permissiveness.

Observation 1. Let $T \geq 1$, $K \in [0,1]$, and r be a rule. $Rel_T(r) \geq K$ iff $Conf(r) \geq K$ and r is not overly permissive with respect to T and K.

We compute the 4-reliability for the rules $Country(\mathbf{u}) = FR$ and $Job(\mathbf{u}) = E$ for the ABAC instance of Figure 1a.

$$
\begin{aligned}
&Rel_4(Country(\mathbf{u}) = FR) \\
&= \min \left\{ \begin{array}{l} Conf(Country(\mathbf{u}) = FR), \\ Conf(Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = E), \\ Conf(Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = M), \\ Conf(Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = S), \\ Conf(Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = T) \end{array} \right\} \\
&= \min\{0.75, 1.0, 1.0, 1.0, 0.0\} \\
&= 0.0.
\end{aligned}
$$

$$
\begin{aligned}
&Rel_4(Job(\mathbf{u}) = E) \\
&= \min \left\{ \begin{array}{l} Conf(Job(\mathbf{u}) = E), \\ Conf(Job(\mathbf{u}) = E \wedge Country(\mathbf{u}) = FR), \\ Conf(Job(\mathbf{u}) = E \wedge Country(\mathbf{u}) = US) \end{array} \right\} \\
&= \min\{0.66, 1.0, 0.5\} \\
&= 0.5.
\end{aligned}
$$

For any measure in Table 2, $Country(\mathbf{u}) = FR$ gets a higher score than $Job(\mathbf{u}) = E$, despite $Country(\mathbf{u}) = FR$ being overly permissive. However, $Rel_4(Country(\mathbf{u}) = FR) < Rel_4(Job(\mathbf{u}) = E)$, as we have shown above.

The previous example and Observation 1 show that reliability achieves what other measures could not: it gives a high score to precisely those rules that have a high confidence and are not overly permissive.

4. Rhapsody

We now present Rhapsody, our ABAC mining algorithm. Rhapsody builds upon APRIORI-SD [15], a machine learning algorithm for subgroup discovery. We start with a brief overview of how APRIORI-SD can be used for ABAC mining.

4.1. APRIORI-SD

APRIORI-SD receives as input two parameters s and c and operates in three stages. We just summarize the main idea and refer the reader to the original paper [15].

1) Compute a set *FreqRules* of rules, such that $r \in$ *FreqRules* iff r covers at least s requests.
2) Compute a subset *ConfRules* $\subseteq$ *FreqRules*, such that $r \in$ *ConfRules* iff $Conf(r) \geq c$.
3) Compute a subset $W \subseteq$ *ConfRules* by iteratively selecting from *ConfRules* the rule with highest weighted relative accuracy (WRAcc on Table 2), until W covers all requests in A.

Although APRIORI-SD mines policies that generalize well, our experiments confirmed that it also mines overly permissive rules. APRIORI-SD uses the WRAcc measure to guide rule selection, which, as discussed in Section 5.2, may give high values to overly permissive rules. For example, when given $s = 4$, $c = 0.5$, and the ABAC instance of Figure 1a, APRIORI-SD outputs the overly-permissive rule $Country(\mathbf{u}) = FR$.

4.2. Rhapsody Algorithm

Rhapsody builds on APRIORI-SD by replacing its last two stages with two new stages. In one of them, Rhapsody computes the reliability of each rule in *FreqRules* and removes those rules whose T-reliability is below a given threshold K. T and K are input parameters defining when a rule is overly permissive. In the other stage, Rhapsody removes those rules that have an equivalent shorter rule. These extensions prevent Rhapsody from mining overly permissive or unnecessarily large rules.

Rhapsody takes as input an ABAC instance (U, P, A, D), $T \geq 1$, and $K \in [0,1]$. Rhapsody outputs an ABAC policy π. A rule r is in π iff it covers at least T requests, $Rel_T(r) \geq K$, and it has no equivalent shorter rule (Theorem 1).

Rhapsody operates in three stages.

1) Compute the set *FreqRules* of all rules covering at least T requests.
2) Compute the subset *RelRules* $\subseteq$ *FreqRules* of rules whose T-reliability is at least K.
3) Remove from *RelRules* all rules that have an equivalent shorter rule in *RelRules* and output the remaining rules.

We explain each stage in detail.

Stage 1 (Algorithm 1). Compute the following:

- A set $FreqRules = \{r : |[\![r]\!]_{U\times P}| \geq T\}$.
- A function $n_{U\times P} : FreqRules \to \mathbb{N}$, such that $n_{U\times P}(r) = |[\![r]\!]_{U\times P}|$.
- A function $n_A : FreqRules \to \mathbb{N}$, such that $n_A(r) = |[\![r]\!]_A|$.

To compute *FreqRules* and $n_{U\times P}$, Rhapsody uses the APRIORI algorithm [16], [18]. We give a brief overview of APRIORI and explain how Rhapsody uses it.

For $s > 0$ and $\mathcal{F}$ a family of sets, we say that a set C is *s-frequent in $\mathcal{F}$* if $|\{S \in \mathcal{F} : C \subseteq S\}| \geq s$. APRIORI receives a family $\mathcal{F}$ of sets and a threshold s and outputs

(i) all s-frequent sets in $\mathcal{F}$ and
(ii) a function $n_{\mathcal{F}}$ mapping each s-frequent set C in $\mathcal{F}$ to $|\{S \in \mathcal{F} : C \subseteq S\}|$.

Rhapsody computes *FreqRules* and $n_{U\times P}$ as follows. First, it computes for each request $(u,p) \in U \times P$ the set $\mathcal{A}(u,p)$ of all atoms that (u,p) satisfies (Line 2). Then it invokes APRIORI on $\{\mathcal{A}(u,p) : (u,p) \in U \times P\}$ with the threshold T (Line 3). Afterwards, it uses APRIORI's output to compute the set *FreqRules* (Line 4). Finally, it computes n_A as follows. Initially, $n_A(r) = 0$, for all $r \in$ *FreqRules* (Lines 5–7). Then, for each $(u,p) \in A$ and each $r \in$ *FreqRules* that (u,p) satisfies, it increments $n_A(r)$ by 1 (Lines 8–12).

Algorithm 1 Rhapsody's first stage

function *Stage1*(U, P, A, T)
  $\mathcal{F} \leftarrow \{\mathcal{A}(u,p) : (u,p) \in U \times P\}$
  $FreqItemSets, n_{U\times P} \leftarrow$ APRIORI($\mathcal{F}, T$)
  $FreqRules \leftarrow$
    $\{\alpha_1 \wedge \ldots \wedge \alpha_k : \{\alpha_1, \ldots, \alpha_k\} \in FreqItemSets\}$
  for $r \in FreqRules$ **do**
    $n_A(r) \leftarrow 0$
  end for
  for $(u,p) \in A$ **do**
    for $r \in FreqRules$ **s. t.** (u,p) satisfies r **do**
      $n_A(r) \leftarrow n_A(r) + 1$
    end for
  end for
  return $FreqRules$, $n_{U\times P}$, n_A
end function

Stage 2 (Algorithm 2). This stage computes the set $RelRules$ of all rules in $FreqRules$ whose T-reliability is at least K.

Definition 8. For two rules r_1 and r_2, we say that r_2 *proves that* $Rel_T(r_1) < K$ if

(i) r_2 is a refinement of r_1,
(ii) $|[\![r_2]\!]_{U\times P}| \geq T$, and
(iii) $Conf(r_2) < K$.

Observe that if a rule proves that $Rel_T(r_1) < K$, then r_1's T-reliability is less than K.

In this stage, Rhapsody computes from $FreqRules$ a subset $UnrelRules$. A rule $r_1 \in FreqRules$ is added to $UnrelRules$ if some rule in $FreqRules$ proves that $Rel_T(r_1) < K$. Afterwards, Rhapsody computes the set $RelRules = FreqRules \setminus UnrelRules$.

Algorithm 2 Rhapsody's second stage

function *Stage2*($FreqRules$, $n_{U\times P}$, n_A, T, K)
  $UnrelRules \leftarrow \emptyset$
  for $r_1, r_2 \in FreqRules$ **do**
    if r_2 proves that $Rel_T(r_1) < K$ **then**
      $UnrelRules \leftarrow UnrelRules \cup \{r_1\}$
    end if
  end for
  $RelRules \leftarrow FreqRules \setminus UnrelRules$
  return $RelRules$
end function

Stage 3 (Algorithm 3). This stage removes redundant rules from $RelRules$ and outputs the result.

Definition 9. We say that a rule r_1 is *equivalent* to a rule r_2 if $[\![r_1]\!]_{U\times P} = [\![r_2]\!]_{U\times P}$.

It is easy to prove that two rules $r_1, r_2 \in RelRules$ are equivalent iff $r_1 \wedge r_2 \in FreqRules$ and $n_{U\times P}(r_1) = n_{U\times P}(r_2) = n_{U\times P}(r_1 \wedge r_2)$.

In this final stage, Rhapsody computes a set $Subsumed$ of redundant rules from the set $RelRules$. For this, each pair of rules $r_1, r_2 \in RelRules$ is analyzed. If r_2 is both shorter than and equivalent to r_1, then r_1 is inserted into $Subsumed$. Afterwards, Rhapsody computes the set $ShortRules = RelRules \setminus Subsumed$.

Algorithm 3 Rhapsody's third stage

function *Stage3*($RelRules, n_{U\times P}, n_A, FreqRules$)
  $Subsumed \leftarrow \emptyset$
  for $r_1, r_2 \in RelRules$ **do**
    if r_1 and r_2 are equivalent and
      r_2 is shorter than r_1 **then**
      $Subsumed \leftarrow Subsumed \cup \{r_1\}$
    end if
  end for
  $ShortRules \leftarrow RelRules \setminus Subsumed$
  return $ShortRules$
end function

Rhapsody's parameters. Rhapsody receives as input two parameters T and K, which guide the selection of mined rules. The values for these parameters depend on the ABAC instance and affect how well the mined policy generalizes. To find the best values, we recommend evaluating Rhapsody as described in Section 5.3 with different values and then choosing those values where Rhapsody mined the policy that generalized better than the others. In our experiments, the best values for T and K are respectively around $0.01 * |U \times P|$ and $|A| \,/\, |U \times P|$.

Rhapsody's performance. Rhapsody's first stage's time complexity is determined by APRIORI's time complexity, which is $O(|U \times P| * L)$, where L is the maximum number of rules that a request may satisfy. Rhapsody's second and third stage's time complexity is quadratic in $|FreqRules| * L$. In the worst case, $|FreqRules|$ is $O\left(|U \times P| * L\right)$. So Rhapsody's time complexity is $O(|U \times P|^2 * L^3)$.

Note that L grows exponentially in the number of attributes, so Rhapsody cannot mine logs with many attributes. Fortunately, in access control scenarios, the number of attributes is usually small, namely less than 20 [29]. Even for the case of the Amazon logs, the number of attributes is less than 15 [20], [21]. This allows Rhapsody to mine from any of the ABAC instances in our experiments within 24 hours. Moreover, there already exists *feature selection* techniques for mining access control policies [22], so one can extract a small subset of attributes that are relevant for deciding authorization, before executing Rhapsody.

In addition, an ABAC mining algorithm does not need to mine policies in real time and organizations specify their access control policies only infrequently. Since the implemented policy is security critical, an offline algorithm providing guarantees is preferable to an online algorithm providing overly-permissive or unnecessarily long rules.

Policy simplification. After executing Rhapsody on (U, P, A, D), it is still possible to prune redundant rules from π by keeping only a small subset that covers all requests covered by π. In our experiments, we found that the best way to build this subset is using APRIORI-SD's last stage. This stage iteratively selects the rule with highest weighted relative accuracy (WRAcc in Table 2) from π until all selected rules cover A. Algorithm 4 gives the details.

Algorithm 4 Policy simplification

```
1: function simplify(π, U, P, A)
2:     uReqs ← U × P
3:     uLog ← A
4:     simpPolicy ← ∅
5:     while uLog ≠ ∅ and π ≠ ∅ do
6:         r ← argmax_{r'}{WRAcc(r') : r' ∈ π}
7:         uReqs ← uReqs \ {(u,p) : (u,p) ⊨ r}
8:         uLog ← uLog \ {(u,p) : (u,p) ⊨ r}
9:         π ← π \ {r}
10:        simpPolicy ← simpPolicy ∪ {r}
11:    end while
12:    return simpPolicy
13: end function
```

Correctness The following theorem establishes Rhapsody's main property: Rhapsody mines exactly those rules that have high reliability and are shorter than any other equivalent rule. The proof is a straightforward consequence of the definitions and algorithms in Section 4.

Theorem 1. A rule r is in Rhapsody's output iff
(i) $|[\![r]\!]_{U\times P}| \geq T$,
(ii) $Rel_T(r) \geq K$, and
(iii) there is no rule r' that is both shorter than and equivalent to r, with $Rel_T(r') \geq K$.

5. Evaluating generalization

We discuss next two common methods for evaluating the precision and generalization of models mined from logs. We argue why they are inadequate and then propose a better method: *universal cross-validation*.

5.1. Limitations of using an organizational security policy for evaluation

One evaluation method for ABAC mining algorithms uses only ABAC instances where the organizational security policy is already known. In this method, a mining algorithm is given the instance as input and the set of requests authorized by the mined policy is compared with the set of requests authorized by the security policy. For example, a policy mined from the instance in Figure 1a, would be compared with the security policy, which is the one described by the light-green shaded rectangles in Figure 1b. We argue next why this approach is *not an adequate way* to evaluate mining algorithms.

In ABAC instances with sparse logs, there are rules covering a significant number of requests, but none of which occurs in the log. We call these rules *uncertain*. In Figure 1a, the rules $Country(\mathbf{u}) = US \wedge Job(\mathbf{u}) = S$ and $Country(\mathbf{u}) = FR \wedge Job(\mathbf{u}) = T$ are uncertain.

Observe that a mining algorithm cannot decide if an uncertain rule is part of the organizational security policy. Therefore, whenever a mining algorithm mines an uncertain rule, there is the risk that the rule is not part of the security policy, and hence, incorrectly authorizes requests. As discussed in Section 2.4, mining algorithms should opt for an incorrect denial in this case. Since the requests covered by the uncertain rule do not occur in the log, this means that these requests happen infrequently, so the work required by the policy administrator in case of an incorrect denial remains low.

Mining algorithms should mine a rule only if the log provides evidence for that rule. Therefore, if an algorithm mines an uncertain rule, then it should be penalized, *even when this rule happens to be part of the organizational security policy*, like the rule $Country(\mathbf{u}) = US \wedge Job(\mathbf{u}) = S$ in Figure 1b.

It is for this reason, that the approach of evaluating mined policies using the organizational security policy is not adequate, as it may not penalize algorithms mining uncertain rules. In contrast, universal cross-validation, which we present in Section 5.3, uses metrics that penalize algorithms mining uncertain rules.

5.2. Limitations of using cross-validation on logs

A standard method for evaluating the precision and the generalization of models mined from logs is cross-validation [30]–[32]. In its simplest form, cross-validation splits the log into a *training* and a *testing log*. Only the training log is given to the algorithm. Once the algorithm finishes, the mined policy is evaluated on the testing log using performance metrics like the *true positive rate (TPR)* and *false positive rate (FPR)* [24]. The true positive rate is the fraction of authorized requests in the testing log that are correctly authorized by the mined policy. The false positive rate is the fraction of denied requests in the testing log that are incorrectly authorized by the mined policy. Figure 2 illustrates cross-validation on logs.

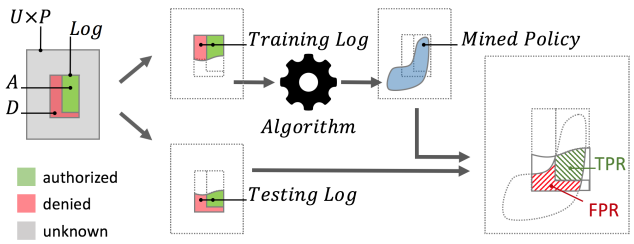
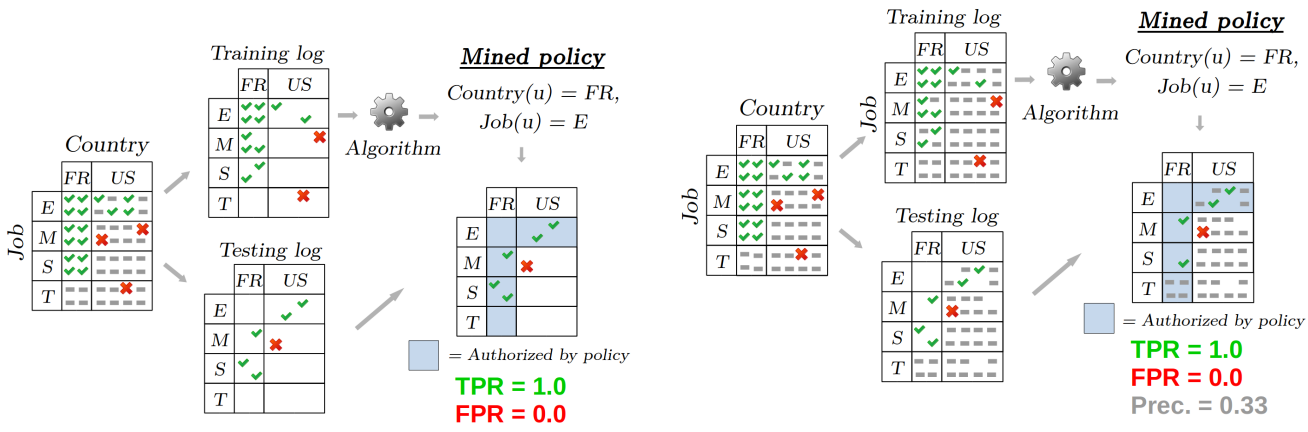


Figure 2: Cross-validation on logs.

If we use cross-validation to evaluate ABAC mining algorithms, then we would give *only the training log* as



(a) In cross-validation on logs, algorithms are given as input only the log $A \cup D$ of an ABAC instance (U, P, A, D), so they cannot evaluate the over-permissiveness of the mined rules. Moreover, the testing log does not include requests outside the original log. In consequence, algorithms mining overly permissive rules, like $Country(\mathbf{u}) = FR$, are *not* penalized.

(b) In universal cross-validation, algorithms are now given as input (U, P, A, D). Moreover, the testing log is expanded with requests not occurring in the original log and the precision of the mined policy is measured. With these changes, algorithms mining overly permissive rules, like $Country(\mathbf{u}) = FR$, are penalized.

Figure 3: Cross-validation on logs and universal cross-validation.

input to the mining algorithm, and then we would evaluate the mined policy *only on the requests in the testing log*. Mining algorithms would not receive in the input requests outside the log and mined policies would not be evaluated on requests not occurring in the log.

In the context of ABAC mining from sparse logs, cross-validation on logs validates overly permissive rules. To illustrate this, we use the ABAC instance from Figure 1a and the classification-tree learning algorithm (CTA) [10], [19], which is used to train classification trees from labeled data. It is easy to extract rules from a classification tree explaining how the tree classifies instances. Hence, CTA is suitable for ABAC mining. We illustrate next what happens if we use cross-validation on logs in this scenario.

Suppose that we split the log into a training and a testing log, as shown in Figure 2. If we use CTA on the training log, then we obtain the policy $\pi = \{Country(\mathbf{u}) = FR, Job(\mathbf{u}) = E\}$, which correctly authorizes and denies all requests in the testing log. Since no French technician has requested access yet, it seems like π perfectly grants and denies access. However, π contains the rule $Country(\mathbf{u}) = FR$ and, as discussed in Section 5.2, this rule is questionable, as it authorizes all French technicians with no evidence from the log. Cross-validation does not evaluate π on those requests because the log does not contain requests from French technicians.

The problem of using cross-validation on logs is that it does not evaluate the *precision* of the mined policy. Precision is a standard machine-learning metric that measures the ratio of authorized requests covered by the policy to the total number of requests covered by the policy [24]. A mined policy may correctly authorize all requests in the log, but it may also have a low precision and authorize an unnecessarily large number of non-logged requests. A good ABAC mining algorithm should be able to anticipate from the log which requests are likely to occur in the future that can also be authorized, but without authorizing requests for which the log gives no evidence.

Although our critique addresses a simple form of cross-validation, it directly extends to more sophisticated forms of cross-validation, like K-fold cross-validation and leave-one-out cross-validation [27].

5.3. Universal cross-validation

Cross-validation on logs is problematic because it ignores all requests that are not in the log. Therefore, to validate an ABAC mining algorithm, we propose to include these ignored requests in the validation process. A good ABAC mining algorithm should then mine policies that fulfill the following properties:

- Attain a high TPR by authorizing as many authorized requests in the testing log as possible.
- Attain a low FPR by authorizing as few denied requests in the testing log as possible.
- Attain a high precision by authorizing as few requests outside the log as possible.

Based on the above ideas, we present *universal cross-validation*, a new method for evaluating ABAC mining algorithms, illustrated in Figure 4. Assume given an ABAC mining algorithm and an ABAC instance (U, P, A, D). Sample uniformly at random two training sets $Tr(A)$ and $Tr(D)$ from A and D, respectively. Let $Ts(A) := A \setminus Tr(A)$ and $Ts(D) := D \setminus Tr(D)$. Give $(U, P, Tr(A), Tr(D))$ as input to the algorithm. Once the algorithm outputs a policy π, evaluate π's *F1 score* and π's *false positive rate* as follows.

$$TPR(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(A)|}{|Ts(A)|}.$$

$$Prec(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(A)|}{|[\![\pi]\!]_{(U\times P)} \cap Tr(A) \cup Tr(D)|}.$$

$$FPR(\pi) = \frac{|[\![\pi]\!]_{U\times P} \cap Ts(D)|}{|Ts(D)|}.$$

$$F1(\pi) = \frac{2 * TPR(\pi) * Prec(\pi)}{TPR(\pi) + Prec(\pi)}.$$

Recall that $[\![\pi]\!]_S$, for $S \subseteq U \times P$ is the subset of requests in S authorized by π. We use the F1 score and the FPR to measure how well a policy generalizes.

Observe that when computing the scores above, all requests have the same weight. However, the policy administrator can add more weight to requests for permissions that are more critical than others, so that policies incorrectly authorizing or denying critical requests are more heavily penalized.

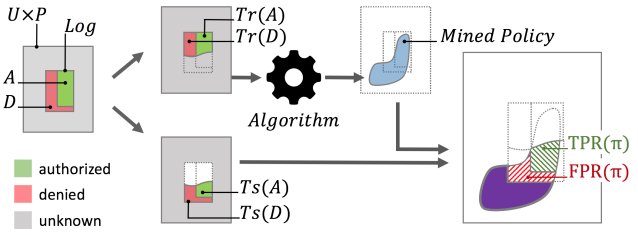


Figure 4: Universal cross-validation.

Figure 3 compares cross-validation on logs with universal cross-validation. First, the input to the algorithm is not only the log, but also the set of all possible requests. Second, universal cross-validation enlarges the set over which algorithms are evaluated. This set now includes a sample from all unknown requests (see the bold purple area in Figure 4).

6. Experiments

In this section, we experimentally demonstrate the four limitations of the state-of-the-art.

Sparse logs. Previous ABAC mining algorithms generalize poorly when mining from sparse logs.

Over-permissiveness. Subgroup-discovery algorithms mine overly-permissive rules.

Rule size. Current machine-learning algorithms mine unnecessarily large rules.

Cross-validation. Cross-validation finds policies that generalize worse than those found using universal cross-validation.

Moreover, we show that Rhapsody is the only algorithm that is capable of mining *succinct* rules from *sparse logs*, without mining *overly permissive* rules.

6.1. ABAC instances

We use ABAC instances from four case studies for our evaluation. We summarize them briefly and refer to Appendix A for details.

Amazon 1. We built four instances from an access log provided by Amazon in Kaggle, a platform for predictive modeling competitions [20], [33]. The log contains more than 12,000 users and 7,000 permissions and is very sparse. For any permission, less than 7% of all users have requested access.

Amazon 2. We built seven instances from another log provided by Amazon in the UCI machine learning repository [21]. The log contains more than 36,000 users and 27,000 permissions. For any permission, less than 10% of all users have requested access.

University. We used a log of students accessing a computer lab in a university. The log contains more than 50,000 users. Less than 1% of the students have requested access and less than 5% of them were denied access. Experiments with these logs were approved by the university's security department.

Basic Organization. We generated five simple synthetic instances, where users and permissions contain only one attribute. All users except those with a specific attribute value are authorized to have any permission. The logs in these instances contain approximately 50% of all possible requests and less than 5% of the logged requests are denied.

6.2. Algorithms

We compare Rhapsody with the following algorithms.

1 Xu and Stoller's ABAC miner [5].
2 CN2 [12], an algorithm for learning classification rules. We used the implementation provided by the Orange data mining library [34].
3 The classification-tree learning algorithm (CTA) provided by the scikit-learn library [35].
4 APRIORI-SD [15], as described in Section 4.

We give an overview of these algorithms in Section 7.

6.3. Evaluation methodology

For each ABAC mining algorithm and each ABAC instance, we ran cross-validation on logs five times and then computed the average FPR, average F1 score, average TPR, and average size of the mined policies. Similarly, we ran universal cross-validation five times and then computed the average of the same metrics. In both types of cross-validation we split the log into a training log and a testing log containing 80% and 20% of the requests, respectively.

Each mining algorithm has a set of parameters that can affect the F1 score, FPR, TPR, and size of the mined policy. We evaluated each algorithm with different values for such parameters. Among all the policies mined by the algorithm, we selected the one with highest F1 score, subject to an FPR < 0.05.

APRIORI-SD and Rhapsody were evaluated on machines with a 2,8 GHz 8-core CPU and 32 GB of RAM. CTA and CN2 were evaluated on machines with a 3,8 GHz 8-core CPU and 32 GB of RAM. All algorithms were given a time limit of 24 hours for each instance. CN2 timed out

when mining the instances from Amazon 2 and University. Xu and Stoller's miner timed out when mining the instances from University.

6.4. Results

Figure 7 compares the F1 score of policies selected using cross-validation on logs with the F1 score of policies selected using universal cross-validation. In most of the cases, the policy selected by universal cross-validation has an FPR equal to 0 (not shown in Figure). Figures 5a, 5b, and 6 compare, respectively, the F1 scores, sizes, and TPRs of the policies mined by each algorithm on the instances of each case study. The FPR was close to 0 for almost all mined policies. From these figures, we make the following four observations.

Cross-validation. In 80% of the cases, the policy selected by universal cross-validation has a higher or equal F1 score than the policy selected by cross-validation on logs. In most of the cases, the improvement is of at least 30%. Observe that the improvement holds for *any ABAC mining algorithm*.

Sparse logs. Xu and Stoller's ABAC miner does not generalize well. In most of the cases, policies mined by Xu and Stoller's attain an F1 score equal to 0. This is because this algorithm was designed mainly for mining from *non-sparse* logs, where a large percentage of all access requests have already been decided. In contrast, Rhapsody's F1 score is among the highest, for almost all instances.

Over-permissiveness. APRIORI-SD mines overly permissive rules. Consider the results for the instances of the basic organization in Figure 6. The TPR of the policies mined by APRIORI-SD is below 0.4 whereas the TPR of the policies mined by Rhaposdy is close to 1. This is because APRIORI-SD uses the confidence measure to mine rules. In Section 3, we explained how this measure leads to mining policies with overly permissive rules, which yielded FPRs above 0.1. The only policies mined by APRIORI-SD that attained FPR < 0.05, as we required in Section 6.3, attained a TPR < 0.4.

Rule size. CTA and CN2 mine unnecessarily large rules. In the Amazon 1 instances, the policies mined by CN2 have size at least twice as large as those mined by Rhapsody and those mined by CTA are 10 times larger than those mined by Rhapsody. This cannot be fixed by expanding CTA or CN2 with Rhapsody's third stage, which searches for each mined rule, the shortest equivalent rule. This is because these algorithms cannot compute for each rule the set of equivalent rules.

We conclude that Rhapsody is the only ABAC mining algorithm capable of mining *succinct* rules from *sparse logs*, without mining *overly permissive rules*. Competing approaches mine rules that generalize poorly, mine unnecessarily large rules, or mine overly permissive rules.

7. Related work

Numerous algorithms have been proposed to mine policies from existing assignments of permission to users or from logs recording which users have required which permissions for their jobs. The approaches taken have been primarily oriented towards role-based access control (RBAC), e.g., [9], [22], [36], and more recently towards ABAC [5], [8], [20]. Moreover, there are machine learning algorithms that learn models from sets of labeled requests, e.g. [9]–[13], [22]. The models learned by these algorithms generalize well and can be adapted to ABAC mining. We discuss them next.

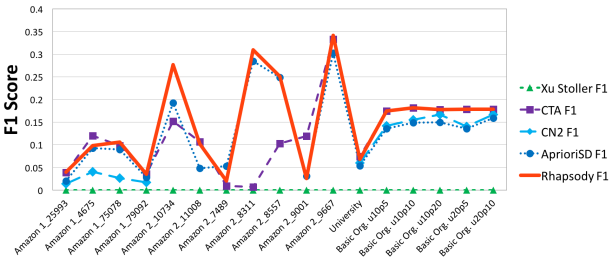
Xu and Stoller's ABAC miner. Recently, Xu and Stoller proposed an ABAC mining algorithm [5], [8]. Their algorithm can mine ABAC policies from access control matrices and logs. However, their algorithm considers only the case of logs that contain a large fraction of the requests. In contrast, Rhapsody can mine succinct policies that generalize well, even from sparse logs.

Rule learning. Rule learning addresses the following problem: Given a set of requests, where each request is labeled as positive or negative, find a set of rules that describe the requests labelled as positive. Several algorithms have been proposed for this, such as CN2 [12] and RIPPER [37]. They all work iteratively, where each iteration learns one rule at a time. In each iteration, the algorithm learns a rule r by computing a series of rules r_0, $r_1, \cdots, r_k$, where $r_0 = \mathbf{true}$, $r_{i+1} = r_i \wedge \alpha_i$, for $i \leq k$, and $r_k = r$. The atom α_i is chosen in a way that r_{i+1} maximizes a rule quality measure. After r_k is computed, all requests that satisfy r_k are removed and the algorithm starts learning another rule. This is repeated until all positive requests are covered or a given termination condition is satisfied.

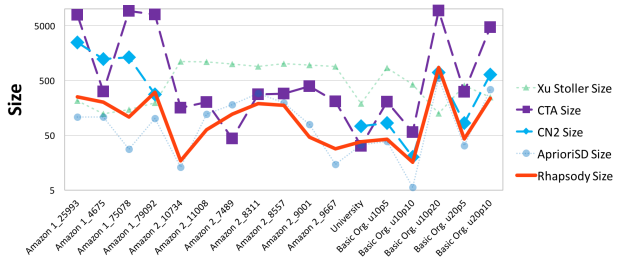
The main limitation of these algorithms stems from their greedy behavior. There may be a high-quality rule where each of its atoms has a low quality, according to the quality measure. These rules will not be mined by the rule learning algorithm. Rhapsody, in contrast, uses ideas from association rule mining [16], [18]. This guarantees that if a high-quality rule is very often satisfied, then Rhapsody will find it, irrespective of the quality of its atoms. Hence, Rhapsody can discover rules that are ignored by rule learning algorithms and can propose more accurate and more succinct rules.

Classification trees. A classification tree is a function encoding a partition of a set of labeled requests. Each partition has an associated label. To predict a value for a new request, the classification tree finds the partition where the new request belongs and uses the label associated to that partition as prediction. Algorithms for mining classification trees [10], [11] yield trees that generalize well. Moreover, one can easily extract rules from those trees. However, as our experiments demonstrated, these rules are unnecessarily long. Rhapsody, in contrast, keeps track of all possible ways to specify a rule and at the end selects the most succinct one.

Random forests and neural networks. Classification trees are prone to overfitting because of their low bias and high variance [27]. For this reason, random forests are rec-



(a) Average F1 score of the policies mined by ABAC mining algorithms. Policies with *higher* F1 score are better as they are more accurate in deciding requests outside the log.



(b) Average sizes of the policies mined by all ABAC mining algorithms. The sizes are logarithmically scaled. *Smaller* policies are better, as they are easier to maintain and audit.

Figure 5: F1 score and size of the policies mined by ABAC mining algorithms.

ommended. A random forest is a collection of classification trees trained over subsamples of the data. The classification decision of a random forests is obtained from the classification decisions of each tree, usually by a majority vote.

Random forests generalize better than classification trees, but this comes at the expense of interpretability [38]. There is no canonical way to extract decision rules from a random forest. For this reason, we cannot apply the algorithms proposed by the winners of the Kaggle competition [39] for ABAC mining. Their models involve mixtures of 15 different models, including classification trees and logistic models. The competition only measured models according to how well the models generalized and not the simplicity of the rules proposed. The requirement of mining simple rules makes other techniques like neural networks unsuitable for ABAC mining.

Subgroup discovery. Subgroup discovery algorithms mine frequent and statistically significant rules from a set of labeled requests. These algorithms [15], [40], [41] require as input a threshold T for the number of requests that must satisfy a rule to be considered as frequent. Moreover, the rules must be statistically significant: the distribution of the requests' labels satisfying this rule must differ significantly from the distribution of all requests' labels. These algorithms compute *all* statistically significant rules that are satisfied by at least T requests.

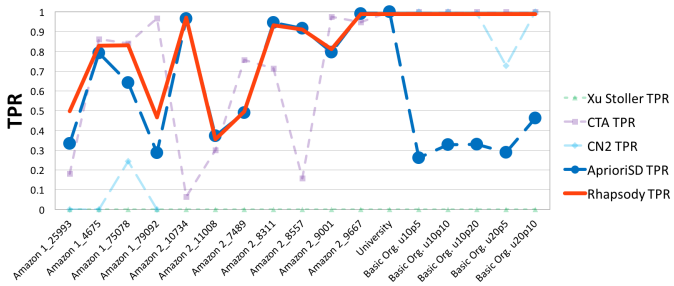


Figure 6: Average TPR of the policies mined by ABAC mining algorithms. Observe that the policies mined by APRIORI-SD have a TPR below 0.4 for the basic organization instances, whereas those mined by Rhapsody have a TPR close to 1.

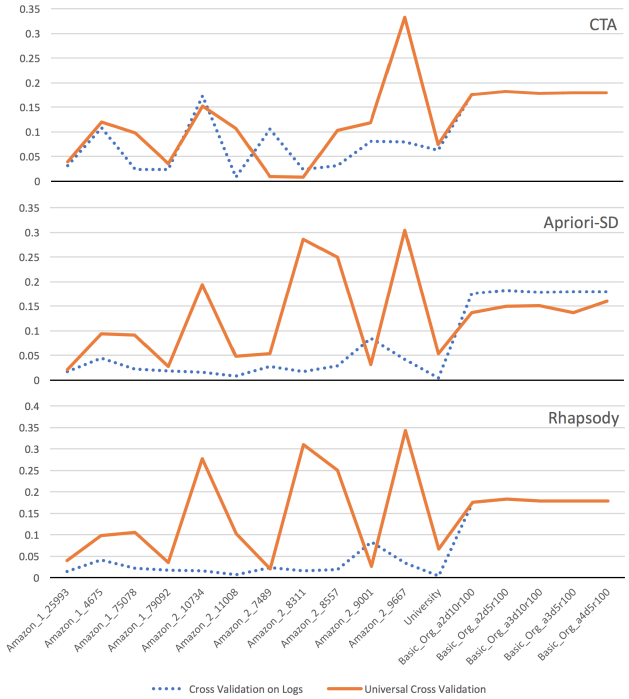


Figure 7: Comparison of F1 score of policies selected using cross-validation on logs versus F1 score of policies selected using universal cross-validation. Policies with *higher* F1 score are better as they are more accurate in deciding requests outside the log.

All subgroup discovery algorithms use rule quality measures that depend just on the confidence of the rule, which can lead these algorithms to mine overly permissive rules. For example, the subgroup discovery algorithm APRIORI-SD uses the WRAcc measure for rule selection. As a result, it mines the overly permissive rule $Country(\mathbf{u}) = FR$ when given as input the ABAC instance of Figure 1a. Our experiments with the basic organization (Section 6.4) also show that APRIORI-SD mines overly permissive rules. In contrast, Rhapsody uses our new measure, reliability, for rule selection. Reliability is guaranteed to select rules with

high confidence that are not overly permissive (Theorem 1).

8. Conclusion

Mining ABAC policies from logs can identify future access requests that should be authorized. To get the maximum benefit from this, one should mine policies before a large fraction of all requests have been decided. When the log contains limited information, we observed two phenomena. First, cross-validation on logs is insufficient as it validates policies with overly permissive rules. Second, state-of-the-art algorithms mine policies with overly permissive rules or unnecessarily large rules.

We proposed universal cross-validation as the method for evaluating mined policies. This method penalizes policies with overly permissive rules but without causing mining algorithms to overfit logs. We also proposed a new measure, reliability, that quantifies better than standard measures how overly permissive a rule is. Based on reliability, we developed a new ABAC mining algorithm, Rhapsody, which guarantees to mine exactly all rules that cover a large number of requests, have a reliability above a given threshold, and have no shorter equivalent rule. When compared with other ABAC mining algorithms, Rhapsody mines policies that generalize better and have smaller size.

As future work, we plan to apply Rhapsody to mine interpretable models from more sophisticated machine-learning models like deep neural networks, gradient boosted decision trees, and random forests.

References

[1] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone *et al.*, "Guide to attribute based access control (abac) definition and considerations (draft)," *NIST special publication*, vol. 800, no. 162, 2013.

[2] National Cybersecurity Center of Excellence, "Attribute-based Access Control," 2017. [Online]. Available: https://nccoe.nist.gov/projects/building-blocks/attribute-based-access-control

[3] G. Kreizman, A. Allan, F. Gaehtgens, B. Iverson, and A. Singh, "Identity and access management scenario 2020: Powering digital business," 2015. [Online]. Available: https://www.gartner.com/doc/3174723/identity-access-management-scenario-

[4] Z. Xu and S. D. Stoller, "Mining attribute-based access control policies from RBAC policies," in *Emerging Technologies for a Smarter World (CEWIT), 2013 10th International Conference and Expo on*. IEEE, 2013, pp. 1–6.

[5] ——, "Mining attribute-based access control policies from logs," in *Data and Applications Security and Privacy XXVIII*. Springer, 2014, pp. 276–291.

[6] S. N. Chari and I. M. Molloy, "Generation of attribute based access control policy from existing authorization system," Sep. 2 2014, US Patent App. 14/474,747.

[7] D. Mocanu, F. Turkmen, and A. Liotta, "Towards ABAC Policy Mining from Logs with Deep Learning," in *Proceedings of the 18th International Multiconference*, ser. Intelligent Systems, 2015.

[8] Z. Xu and S. D. Stoller, "Mining attribute-based access control policies," *Dependable and Secure Computing, IEEE Transactions on*, vol. 12, no. 5, pp. 533–545, 2015.

[9] I. Molloy, Y. Park, and S. Chari, "Generative models for access control policies: applications to role mining over logs with attribution," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. ACM, 2012, pp. 45–56.

[10] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[11] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[12] P. Clark and T. Niblett, "The CN2 induction algorithm," *Machine learning*, vol. 3, no. 4, pp. 261–283, 1989.

[13] J. Hühn and E. Hüllermeier, "FURIA: an algorithm for unordered fuzzy rule induction," *Data Mining and Knowledge Discovery*, vol. 19, no. 3, pp. 293–319, 2009.

[14] V. Jovanoski and N. Lavrač, "Classification rule learning with apriori-c," in *Portuguese Conference on Artificial Intelligence*. Springer, 2001, pp. 44–51.

[15] B. Kavšek and N. Lavrač, "Apriori-SD: Adapting association rule learning to subgroup discovery," *Applied Artificial Intelligence*, vol. 20, no. 7, pp. 543–583, 2006.

[16] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th International Conference on Very Large Databases, VLDB*, vol. 1215, 1994, pp. 487–499.

[17] N. Lavrač, P. Flach, and B. Zupan, *Rule evaluation measures: A unifying view*. Springer, 1999.

[18] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *ACM SIGMOD Record*, vol. 22, no. 2. ACM, 1993, pp. 207–216.

[19] Scikit-learn, "sklearn.tree.DecisionTreeClassifier," 2013. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier

[20] Kaggle, "Amazon.com – Employee access challenge," 2013. [Online]. Available: http://www.kaggle.com/c/amazon-employee-access-challenge

[21] M. Lichman, "UCI machine learning repository. amazon access samples data set," 2013. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Amazon+Access+Samples

[22] M. Frank, A. P. Streich, D. Basin, and J. M. Buhmann, "A probabilistic approach to hybrid role mining," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 101–111.

[23] Orange, "Orange: Rule Induction with CN2," 2013. [Online]. Available: http://orange.readthedocs.io/en/latest/reference/rst/Orange.classification.rules.html

[24] D. M. Powers, "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation," 2011.

[25] X. Jin, R. Krishnan, and R. Sandhu, "A unified attribute-based access control model covering dac, mac and rbac," in *Data and Applications Security and Privacy XXVI*. Springer, 2012, pp. 41–55.

[26] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-based access control." *IEEE Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[27] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer Series in Statistics, Berlin, 2001, vol. 1.

[28] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[29] I. Molloy, J. Lobo, and S. Chari, "Adversaries' holy grail: access control analytics," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM, 2011, pp. 54–61.

[30] A. D'yakonov, "Solution methods for classification problems with categorical attributes," *Computational Mathematics and Modeling*, vol. 26, no. 3, pp. 408–428, 2015.

[31] Q. Yang, H. H. Zhang, and T. Li, "Mining web logs for prediction models in WWW caching and prefetching," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 473–478.

[32] M. de Leoni and W. M. van der Aalst, "Data-aware process mining: discovering decisions in processes using alignments," in *Proceedings of the 28th annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1454–1461.

[33] Kaggle, "Kaggle: the home of data science," 2017. [Online]. Available: http://www.kaggle.com

[34] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[36] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, "Mining roles with semantic meanings," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. ACM, 2008, pp. 21–30.

[37] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the twelfth international conference on machine learning*, 1995, pp. 115–123.

[38] L. Song, P. Langfelder, and S. Horvath, "Random generalized linear model: a highly accurate and interpretable ensemble predictor," *BMC bioinformatics*, vol. 14, no. 1, p. 1, 2013.

[39] Kaggle, "Amazon.com – Employee access challenge. Winners' solution and final results." 2013. [Online]. Available: https://www.kaggle.com/c/amazon-employee-access-challenge/forums/t/5283/winning-solution-code-and-methodology

[40] M. Atzmueller and F. Puppe, "SD-map–a fast algorithm for exhaustive subgroup discovery," in *Knowledge Discovery in Databases: PKDD 2006*. Springer, 2006, pp. 6–17.

[41] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski, "Subgroup discovery with CN2-SD," *The Journal of Machine Learning Research*, vol. 5, pp. 153–188, 2004.

Appendix A.
ABAC instances used for experiments

Amazon 1. These are instances built from two access logs provided by Amazon in Kaggle, a platform for predictive modelling competitions [20], [33]. One log is for training and contains access requests made by Amazon's employees over two years [20]. Each entry in this log describes an employee's request to a resource and whether the request was authorized or not. The request contains all the employee's attribute values and the resource identifier. The second log is for evaluation. It contains access requests only, but it does not specify which requests are authorized. Participants in the Kaggle competition had to decide for the evaluation log which requests to authorize. The logs contain more than 12,000 users and 7,000 resources.

From the Amazon logs one can build an ABAC instance (U, P, A, D), where U and P are the set of users and the set of resources occurring in the logs, respectively, and $A \cup D$ are the requests occurring in the training log. However, such an instance is too large to fit in main memory and some of the implementations of competing ABAC mining algorithms (Rhapsody included) cannot handle such amounts of data. To deal with this, we observe that the only permission attribute is the resource's identifier, so any ABAC rule authorizes requests for at most one resource. Therefore, any ABAC policy for (U, P, A, D) can be partitioned into several policies, each authorizing requests for only one resource. Hence, rather than mining over (U, P, A, D), we can mine over instances of the form $(U, \{p\}, A_p, D_p)$, where p is a single resource and $A_p \cup D_p$ are all requests for p occurring in the training log. These instances are much smaller and are easily handled by all competing ABAC mining algorithms.

For our experiments, we selected the five instances $(U, \{p\}, A_p, D_p)$ with the highest value for $|A_p \cup D_p|$. In all cases, $|A_p \cup D_p| \,/\, |U| < 0.07$. Hence, the log contains, for each resource, less than 7% of all possible requests.

Amazon 2. These are instances built from access data provided by Amazon in the UCI machine learning repository [21]. The access data contains more than 36,000 users and 27,000 permissions. We took the eight most requested permissions and for each of them, we created an ABAC instance $(U, \{p\}, A, D)$ where U are all users in the access data, p is the permission, A are the users who requested p and were authorized and D are those who requested p and were denied.

Basic Organization. These are synthetic instances with only one user attribute value, Job, identifying the job the user performs and only one permission attribute value, $Category$, identifying the category where the permission belongs. We use natural numbers to identify jobs and categories. There is only one permission for each category and there are 100 users for each job. For each category, we assume that all jobs except one are authorized to request permissions for that category. For each category c, we denote by p_c the permission from that category and by j_c the job that is *not* authorized to request p_c. We let J and C denote the total of jobs and categories, respectively. The values for J and C in each instance are described in Table 3.

TABLE 3: Properties of the basic organization policies

	Instance				
	1	2	3	4	5
Num. jobs.	10	10	10	20	20
Num. categories.	5	10	20	5	10

We now describe the log for each instance. For each category c and for each job $j \neq j_c$, the fraction of users with job j that have requested p_c is c/C. This is just to simulate a non-uniform distribution of the categories of the permissions requested by users. In addition, for each category c, there is only one user with job j_c that has requested access to p_c. This is just to ensure fewer denied requests than authorized requests in the log.