Automating Audit with Policy Inference

Abhishek Bichhawat  
*Carnegie Mellon University*  
Pittsburgh, USA  
abichhaw@andrew.cmu.edu

Matt Fredrikson  
*Carnegie Mellon University*  
Pittsburgh, USA  
mfredrik@cs.cmu.edu

Jean Yang  
*Akita Software, Inc.*  
San Mateo, USA  
jean@akitasoftware.com

Abstract—The risk posed by high-profile data breaches has raised the stakes for adhering to data access policies for many organizations, but the complexity of the policies themselves and the applications that must obey them raises significant challenges. To mitigate this risk, fine-grained audit of access to private data has become common practice, but this is a costly, time-consuming, and error-prone process.

We propose an approach for automating much of the work required for fine-grained audit of private data access. Starting from the assumption that the auditor does not have an explicit, formal description of the correct policy, but is able to decide whether a given policy fragment is partially correct, our approach gradually infers a policy from audit log entries. When the auditor determines that a proposed policy fragment is appropriate, it is added to the system’s mechanized policy, and future log entries to which the fragment applies can be dealt with automatically. We prove that for a general class of attribute-based data policies, this inference process satisfies a monotonicity property which implies that eventually, the mechanized policy will comprise the full set of access rules, and no further manual audit is necessary. Finally, we evaluate this approach using a case study involving synthetic electronic medical records and the HIPAA rule, and show that the inferred mechanized policy quickly converges to the full, stable rule, significantly reducing the amount of effort needed to ensure compliance in a practical setting.

I. INTRODUCTION

Modern organizations and systems, such as e-commerce websites, healthcare systems, banking and financial systems, and online social-networks, are huge applications that handle large amounts of user data. The data is normally subject to different policies based on the user’s preferences and the confidentiality level of the data. Although the consequences of data breaches are severe [1], not all relevant policy checks are enforced in these systems often causing major data leaks [2].

Various factors affect the correct enforcement of policies in large systems — (1) constant modification to the policies, (2) the complexity of the policies, (3) the enormous size of the codebase, which runs into hundreds of thousands of lines of code involving multitude of developers, and (4) continuously evolving codebases resulting in newer policies. These factors make it difficult for developers to ensure that the policy checks are implemented correctly in the applications.

Importantly, most organizations do not have an explicit, mechanized specification of the target policy, either during development or when the system is deployed. While developers might add appropriate checks *inline* with the application code, it is easy to miss necessary checks [3]. The decentralized nature of this type of enforcement makes it difficult to obtain a comprehensive statement of the policy enforced by the system, and thus to identify and fix authorization bugs when they arise. To address the issue of policy compliance, most organizations adopt the approach of auditing transactions and interactions of the users with their systems [4]. The idea is to log different operations performed on the system, not limited to the handling of private user data, and then perform an after-the-fact validation of those transactions. The goal of the validation process is to identify all violations of the organizational policy and, in turn, the violators [2].

Furthermore, auditing transactions is required by certain privacy laws [1, 5, 6] — e.g., the Health Insurance Portability and Accountability Act (HIPAA) [1] requires healthcare organizations to record all activities pertaining to the protected health information of patients, and to perform a timely audit of those transactions for ensuring privacy compliance.

Normally, organizations employ a privacy champion or an auditor to validate the transactions against the laws and the organizational policy. The top half of Figure 1 shows the standard auditing procedure followed by organizations [7]. The auditor understands the organizational policy and laws that govern the handling of data, and validate the logs generated by the system against the policy and laws. Apart from being time-consuming and costly [8], the enormous size of the logs and the number of transactions makes this job both cumbersome and error-prone [9].

The main insight that this work is based on is that the information present in audit logs, when combined with the decisions made by an auditor, is in many cases sufficient.
to reconstruct a correct mechanized policy. If incremental portions of this policy can be audited with approximately as much effort as the log entries they are generated from, then the cost of manual audit can be gradually reduced as more of the mechanized policy is inferred. To this end, we present an approach for automating the auditing process by inferring access policies from log entries, without requiring any prior policy specification.

The bottom half of Figure [1] presents an overview of our approach. As the logs normally contain enough information for the auditor to determine the legitimacy of a transaction, we leverage this information to generate a partial policy, which can subsequently be applied to audit future log entries. As the policy is constructed, the auditor validates each new policy fragment instead of validating individual log entries from which they are generated. Entries that cannot be resolved using the information can be logged for further manual review. Additionally, the mechanized policy can be used to enforce runtime policy compliance, replacing inline policy checks where appropriate.

Our main contribution is a framework for policy inference and automated auditing using logs. We describe a general algorithm that infers the policy from logs and checks the log entries for policy compliance. The policy is represented in a restricted fragment of authorization logic [10, 11], which is a logic for access control. The algorithm employs certain attributes of the log entries to infer abstract formulas from them, and an oracle to validate the inferred policy. Unlike machine learning based mining approaches, our approach does not allow over-permissive policy. The log entries are audited against the validated formulas, and the violations are reported. Instead of requiring the policy to be specified against which the logs are audited, the inference algorithm works with a possibly empty set of initial formulas.

We prove the following properties of the algorithm — (1) the generated policy is monotonically increasing, i.e., it is at least as permissive as the original policy that was input to the algorithm, and eventually reaches a fixed-point; (2) every iteration of the algorithm terminates; (3) the algorithm is sound, i.e., the log used to generate the policy satisfies the policy; (4) the algorithm generates minimal policy: the generated policy is the most restrictive policy that the log satisfies - this property shows that our algorithm does not generate over-permissive policy.

We implement the algorithm to infer policy and detect violations in the audit log and evaluate the algorithm to answer three questions — (1) is auditing policy easier than auditing logs, i.e., are the number of formulas less than the number of log entries and how does this ratio vary for different number of log entries; (2) is the violation detector able to detect violations of access policy in the log using the formulas that fail the policy audit; (3) how expressive and permissive are the formulas as compared to the policy (sans temporal operators). We evaluate our implementation on sets of realistic simulated audit logs in a healthcare setting (governed by HIPAA) by controlling the probability of violation in the logs. We show that the inference algorithm considerably reduces the effort of an auditor by significantly reducing the number of entries that need to be reviewed; the auditor needs to validate around 400 formulas which are only 0.112% of about 350,000 log entries. Additionally, we show that our algorithm detects violations effectively: from 8 incorrect formulas, the violation detector is able to report around 29,000 unauthorized transactions in the log. The formulas inferred by our algorithm are expressive enough to cover most of the HIPAA policy except the clauses with temporal operators.

To the best of our knowledge, this is the first work that presents the idea of automated auditing using policy inference. Existing work either focuses on automated auditing or policy mining but not both. In the area of automated auditing, prior works require explicit policy specification against which the logs are tested for compliance [12,17]. However, policy specification is itself a complex and tedious task [18, 19], and requires the developer to understand the policy language to be able to specify and check the policy that should be enforced by the system. While policy mining from logs have also been well-studied [20–31], prior works employ machine learning algorithms to generate the policy, which might lead to incorrect authorizations.

The rest of this paper is organized as follows. In Section II we present a more detailed overview of our system, and the various components needed to realize our approach. Section III describes the policy logic used by our system, formalizing its syntax and semantics. In Section IV we describe our approach for inferring policies from audit logs, and describe several properties that establish its correctness and usefulness. Section V describes our prototype implementation of the approach, and Section VI details our case study application of it to synthetic electronic medical records. We discuss how various operators are interpreted in our approach, the limitations of our approach and possible future directions in Section VII. Section VIII discusses related work, and Section IX concludes the paper. Proofs of important claims are provided in the appendix.

II. System Overview

Organizations have a group of users that perform certain actions on some object or resources in their servers through different applications. The resources are organized based on their type; in the context of a database, a column in a table (or the table itself) can represent the type of the data it stores. For instance, a profile table may have the columns NAME and ADDRESS, both of which might be separate types or could be represented together as PERSONAL INFORMATION.

Every user has fixed roles that normally define the user's access properties while the various endpoints of an application define the type of information accessed. A user can have multiple roles in the system, e.g., a user can be both a nurse and a researcher in a healthcare system, but every transaction performed by the user is specific to a role. Applications normally use credentials or some state, e.g., cookies, to differentiate various logins by a user. The users and resources,
additionally, have specific attributes that might affect how a user accesses some resource. These attributes describe certain property of the user or the resource. Role of a user, for example, may be one of the attributes that decides access policies for the user. Attributes are used in attribute-based access control mechanisms to determine whether a user may access a resource.

There are associations and relationships between different users, and between users and resources in the system that further define access policies. For instance, if Alice is a doctor of Bob, she may access the diagnosis information of Bob. Similarly, Eve may be allowed to view the diagnosis of Bob if Eve is a relative of Bob who has been authorized to view the diagnosis. Such associations are an important component when determining access in relationship-based access control mechanisms. Broadly, the system we consider allows both attribute- and relationship-based access control policies to be specified and enforced.

Systems maintain logs of transactions recording activities of the users at various times, which the organizations can use for the purpose of auditing. Auditing is useful in determining violations of organizational (privacy) policies or to detect illicit activity by any of the users in the system [4]. Audit logs do not have a standard format or representation across organizations but normally contain the following information — the action performed, the user performing the action, data or resource accessed and the state of the system at the time of access, and any additional information or attributes depending on the type of transaction and the system. The log \( \mathcal{L} \) contains a list of entries for all actions. Depending on the application, the logs also may include entries for data sent out on the network. In the system we consider, a log entry \( \gamma \) is a tuple of the form: \( \langle A, U, o, \text{einfo} \rangle \) where \( A \) is the action, \( U \) is the user, \( o \) is the resource, \( \text{einfo} \) is any additional information, and \( \tau \) is the time at which the transaction happened.

III. POLICY LOGIC

We represent the policies in a subset of first-order intuitionistic logic [14]. The policies in our system are based on general attributes of users and resources, and are not specific to individual users.

A. Policy Syntax

Figure 2 shows the syntax of policies. Terms \( (t) \) represent both users and resources in the system. In fact, the domain of terms includes all the entities and at least the principals involved in the system. The type of an entity (term) in the system is represented as \( \sigma \), e.g., principal represents all individuals associated with the system who are authorized access and define the policies. The domain of every type in the system is fixed and bounded.

Actions \( (C) \) correspond to the activities performed by a user on a resource. State predicates \( (P) \) depend on the state of the system entities, either at the current time or in the specified time interval. Atoms \( (a) \) are state predicates applied to a list of terms. Access predicates \( r \) apply an action \( C \) to a list of terms stating that the action was performed at time \( T \). The connective \( \land \) is standard while \( a \supset \text{may} r \) indicates that the access \( r \) can only happen if \( a \) holds. The quantifier \( \forall x \) ranges over all possible terms of a particular type \( \sigma \) if \( x \) has the type \( \sigma \). As the domain of every type \( \sigma \) is bounded in our system, \( \forall x \) ranges only over a finite domain; hence, the number of formulas in the policy are also bounded and finite. To represent a vector of variables of different types, we use \( \vec{x} \). A formula \( s \) has a specific form in our framework and is defined for all terms \( \vec{t} \) in atoms \( a_1 \) and the access predicate \( r \). The policy \( \varphi \) is represented in disjunctive normal form over all formulas \( s_j \).

The formulas can be reduced to define a minimal policy. As all properties and predicates are true only during certain time intervals, we have time-dependent predicates in our logic. The instantiated predicates are time-dependent as shown in the rules in Figure 3.

B. Semantics

The semantics of policy enforcement is defined in Figure 3.

The formal relation \( \Sigma; R; \mathcal{L} \models \varphi \) states that a log \( \mathcal{L} \) satisfies the policy \( \varphi \) under the typing environment \( \Sigma \) and the set of relations \( R \) (derived from a database). We describe the rules in the semantics below.

Rules IND and BASE correspond to the inductive and base case for iterating over a list of log entries in the log. Rule BASE states that an empty log satisfies any policy \( \varphi \) while rule IND states that if a log entry \( \gamma \) satisfies the policy \( \varphi \) and a log \( \mathcal{L} \) satisfies \( \varphi \), then the larger log \( \gamma, \mathcal{L} \) also satisfies \( \varphi \). The rule ACCESS states that a log entry \( \gamma \) satisfies a policy \( \varphi \), if it satisfies any one of the formulas in \( \varphi \). Rule INST states that if \( \gamma \) satisfies a formula \( s \) instantiated with terms in the log entry \( \gamma \), it also satisfies the general formula \( s \). The rule PRED says a log entry satisfies an instantiated formula \( \gamma \), if the state predicates at the time of log entry hold. It uses the judgement \( \Sigma; R; T \vdash a \) to imply that at time \( T \), under the typing environment \( \Sigma \) and the set of relations \( R \), \( a \) holds. The rules OWN, RELN and ATTR return the respective relations of ownership, other relationships, and attributes from the set of relations \( R \) (database) associated with the terms in the
Given two policies \( \varphi_1 \) and \( \varphi_2 \), we say that \( \varphi_1 \leq \varphi_2 \) for all logs \( L \), typing environments \( \Sigma \) and sets of relations \( R \), if \( \Sigma; R; L \models \varphi_1 \), then \( \Sigma; R; L \models \varphi_2 \)

**Lemma 1.** Given two policies \( \varphi_1 \) and \( \varphi_2 \), and a formula \( s \), if \( \varphi_1 \leq \varphi_2 \), then \( \varphi_1 \cup \{ s \} \leq \varphi_2 \cup \{ s \} \)

**Lemma 2.** Given a policy \( \varphi \), for all logs \( L \), typing environments \( \Sigma \) and sets of relations \( R \), if \( \Sigma; R; L \not\models \varphi \), then \( \forall \gamma \in L, \exists s \in \varphi, \Sigma; R; \gamma \not\models \{ s \} \)

**IV. Policy Inference**

Our main contribution is the inference system that builds a policy from scratch given a list of log entries while assisting in the audit of those log entries. Figure 4 describes our policy inference framework. The workflow starts with a set of logs and possibly empty set of formulas input to the policy checker and generator. For all entries in the log, the policy checker checks if the entry satisfies an existing formula in the policy or not. If it satisfies a formula, they are accepted as valid. If not, it generates new formulas for all such entries, and logs those entries in a separate review log. An oracle then reviews the set of generated formulas and decides which of those formulas are allowed as part of the policy and which of those are possible violations. The oracle is a function that takes a policy and returns a valid policy. The review logs are then checked against the oracle approved policy and the violations are then reported for further review. The approved policy is fed back to the system and becomes part of the policy that the policy checker takes as input.
Input: $R$ - list of relations
   $\vec{t}$ - list of variables
   $T$ - time
Output: $P$ - set of predicates

Function `infer $(R, \vec{t}, T)$`:

```plaintext
1  P ← \{\};
2  for i ← 0 to |R| do
3    r ← R[i];
4      if not ownerTuple(r) and
5        r = (id = t1, p = v, (T1, T2)) then
6          if t1 ∈ $\vec{t}$ and $T1 \leq T \leq T2$ then
7            P ← P ∪ \{has_attr t1 p v\};
8        end
9      else if r = (id = (t1, t2), reln = v, (T1, T2)) then
10     if t1 ∈ $\vec{t}$ and t2 ∈ $\vec{t}$ and $T1 \leq T \leq T2$ then
11        P ← P ∪ \{has_reln t1 t2 v\};
12    end
13 end
```

Algorithm 2: Definition of `rel$(R, \vec{t}, T)$`

Input: $L$ - Log
   $R$ - Relations
   $\Sigma$ - typing environment
Output: $\varphi$ - policy

Function `infer $(L, R, \Sigma)$`:

```plaintext
1  \varphi ← \{\};
2  for i ← 0 to |L| do
3    \(r(\vec{t})@T \leftarrow L[i]\);
4    \(P_o, \vec{t} \leftarrow own(R, \vec{t}, T)\);
5    \(P_r \leftarrow rel(R, \vec{t}, T)\);
6    \(P ← P_o ∪ P_r\);
7    \(\vec{x} ← \Sigma(\vec{t})\);
8    s ← (\bigwedge_{p \in P} \varphi[p]) \supset may r(\vec{t})\);
9    s(\vec{x}) ← s(\vec{x}/\vec{t})\);
10   \varphi ← \varphi \cup \{\forall \vec{x}. s(\vec{x})\};
11 end
```

Algorithm 3: Definition of `infer$(L, R, \Sigma)$`

A. Inference Algorithm

The core of our inference algorithm consists of two computable functions — `infer` and `reduce`. The function `infer` generates a policy based on a list of log entries while the `reduce` function takes in two formulas $s_1$ and $s_2$, and returns whether $s_2$ is more permissive than $s_1$, i.e., $\{s_1\} \leq \{s_2\}$. We describe these functions in more detail below.

1) Inference: Algorithm 3 defines the function `infer` that infers a policy from the log entries. It takes as input the log entries ($L$) along with the set of relations in the database $R$ and the typing environment $\Sigma$. The inductive definition generates a set of inferred formulas taking into account the attributes and associations of the users and resources involved in a transaction. The formulas are generalized to include all terms that can satisfy the inferred predicates.

The function `infer` traverses every log entry and generates a formula for it. If the log is empty, it returns an empty policy (line 1). For every log-entry of the form $r(\vec{t})@T$, `infer` (line 2) generates atomic predicates from the set of relations $R$ (lines 4 and 5), using functions `own` (Algorithm 1) and `rel` (Algorithm 2), taking into account the owner of the resource. The returned predicates (line 6) are included as part of the formula (line 8), which is generalized for all variables $\vec{x}$ (line 9) such that $\vec{x}$ has the same type as the set of actual terms $\vec{t}$.

The function `own` (Algorithm 1) iterates through the relations in $R$ and returns the state predicates for ownership of all variables in $\vec{t}$ valid at time $T$. In particular, it returns the state predicate `owner t t'` for all $t$ in $\vec{t}$ if $t'$ is the owner of $t$ at time $T$ (line 6); if not, it returns an empty predicate set (line 1). The meta-function `ownerTuple` used in the function definition (line 3) checks if the tuple $r$ defines the ownership attribute of a resource with a principal. If not, it returns no predicate for that relation. This is required to distinguish ownership attribute with other attributes. The function `owner` importantly, retrieves the owner of a resource in $\vec{t}$ (line 7) to correctly generate all predicates of the formula.

Using the set of terms $\vec{t}$, the function `rel`, defined in Algorithm 2, returns the relationships and attributes for every combination of the terms in $\vec{t}$. The function `rel` iterates over the relations in $R$, and either generates a predicate with terms or returns an empty set of atoms. If the relation describes the attribute of a term (other than ownership) in $\vec{t}$ at time $T$, line 6 generates a `has_attr` predicate with the term, and the corresponding attribute type and value. If the relation describes the relationship between two terms in $\vec{t}$ at time $T$, line 10 generates a `has_reln` predicate with the terms, and the corresponding relationship. If neither conditions hold, `rel` returns the empty set.

As the number of formulas generated are dependent on the attributes of resources and users, using user and resource attributes with unbounded values may result in complex formulas. We address this issue by considering only those attributes that the developer specifies are a part of the policy. For instance, while zipcode of a user might be a possible policy attribute, the complete address might not be one.

a) Example: Consider the following instance of an accepted clause in HIPAA about protected health information where a surgeon Alice sends her patient Bob’s protected health information to Charlie, Bob’s physician, for the purpose of treatment at time $T$. The transaction is logged as:

```
send Alice Bob PHI Charlie purpose:treatment @ T (T1)
```
Suppose

\[ R = \{ (\text{id} = \text{Alice}, \text{role} = \text{doctor}, (T_1, T'_1)); \]
\[ (\text{id} = \text{Charlie}, \text{role} = \text{doctor}, (T_2, T'_2)); \]
\[ (\text{id} = \text{Bob}, \text{role} = \text{patient}, (T_3, T'_3)); \]
\[ (\text{id} = \text{Bob}_{PHI}, \text{owner} = \text{Bob}, (T_4, T'_4)); \]
\[ (\text{id} = \text{Alice}, \text{Bob}), \text{reln} = \text{doctor_of}, (T_5, T'_5)); \]
\[ (\text{id} = \text{Charlie}, \text{Bob}), \text{reln} = \text{doctor_of}, (T_6, T'_6)) \]

where, \( T_i \leq T \leq T'_i \) for \( i = 1 .. 6 \). We have \( \vec{t} = \{ \text{Alice, } \text{Bob}_{PHI}, \text{Charlie} \} \). From own, we have the following predicate:

\[ P_1 = \text{owner } \text{Bob}_{PHI} \text{ Bob} \quad \text{(P1)} \]

The set \( \vec{t} \) now includes Bob, i.e.,

\[ \vec{t} = \{ \text{Alice, Bob}_{PHI}, \text{Charlie, Bob} \} \]

From rel (line 6), we have the following predicates:

\[ P_2 = \text{has_attr } \text{Alice role doctor} \quad \text{(P2)} \]
\[ P_3 = \text{has_attr } \text{Charlie role doctor} \quad \text{(P3)} \]
\[ P_4 = \text{has_attr } \text{Bob role patient} \quad \text{(P4)} \]

and from rel (line 10), we have the following predicates:

\[ P_5 = \text{has_reln } \text{Alice Bob doctor_of} \quad \text{(P5)} \]
\[ P_6 = \text{has_reln } \text{Charlie Bob doctor_of} \quad \text{(P6)} \]

The formula \( s_1 \) inferred by infer might be related such that one is strictly more permissive than the other, we can reduce them to a simpler formula based on the terms involved in the formula. Intuitively, this corresponds to the weakening rule in logic, i.e., if \( A \models C \), then \( A \wedge B \models C \).

We define the function reduce for reducing inferred formulas. The function is defined in Algorithm 4. It takes as arguments two formulas \( s_1 \) and \( s_2 \) along with the typing environment \( \Sigma \), and returns a Boolean value to indicate whether \( s_2 \) is more strict than \( s_1 \).

The function bvar on line 2 initializes \( \vec{x} \) and \( \vec{y} \) with the bounded variables in \( s_1 \) and \( s_2 \) while the functions antt and consq on lines 3 and 4 respectively return the antecedents and consequents of the formulas \( s_1 \) and \( s_2 \). The function checks (on line 5) if the consequents (access predicates) \( r_s(\vec{x}_s) \) and \( r_{sy}(\vec{y}_{sy}) \) of the two formulas are the same under the typing environment \( \Sigma \), and continues if \( \vec{x}_s \) is equal in length and type to \( \vec{y}_{sy} \) (line 6). The abstract function comb on line 9 generates various orderings of the variables in \( \vec{x} \) based on the typing information of the variables in \( \vec{y} \) to get a one-to-one mapping between variables in \( \vec{x}_i \) to variables in \( \vec{y}_i \). Then, reduce substitutes the variables in the set of formulas \( \vec{s} \) with the variables from \( \vec{r} \) (line 10), and if the predicates generated with substitutions are a superset of the predicates in antecedent of \( s_1 \) (line 11), returns True (line 12). If none of the above branch conditions are satisfied, reduce returns False (line 1).

(a) Example: Consider an extension of the example described above where Alice sends Dave’s protected health information to Charlie for the purpose of treatment at time \( T \) Alice is Dave’s doctor and Charlie is not Dave’s physician. The transaction is logged as:

\[ \text{send Alice } \text{Dave}_{PHI} \text{ Charlie purpose:treatment } \boxcirc \text{T'} \quad \text{(T2)} \]

Suppose

\[ R = \{ (\text{id} = \text{Alice}, \text{role} = \text{doctor}, (T_7, T'_7)); \]
\[ (\text{id} = \text{Charlie}, \text{role} = \text{doctor}, (T_8, T'_8)); \]
\[ (\text{id} = \text{Dave}, \text{role} = \text{patient}, (T_9, T'_9)); \]
\[ (\text{id} = \text{Dave}_{PHI}, \text{owner} = \text{Dave}, (T_{10}, T'_{10})); \]
\[ (\text{id} = \text{Alice}, \text{Dave}), \text{reln} = \text{doctor_of}, (T_{11}, T'_{11})) \]

where, \( T_i \leq T' \leq T'_{i'} \) for \( i = 7 .. 11 \). We have \( \vec{t} = \{ \text{Alice, } \text{Dave}_{PHI}, \text{Charlie} \} \). From own, we have the following predicate:

\[ P_7 = \text{owner } \text{Dave}_{PHI} \text{ Dave} \quad \text{(P7)} \]
The set \( t' \) now includes Dave, i.e.,

\[
\tilde{t} = \{ \text{Alice, Dave\_PHI, Charlie, Dave} \}
\]

From rel, we have the following predicates:

- \( P_8 = \text{has\_attr Alice role doctor} \) (P8)
- \( P_9 = \text{has\_attr Charlie role doctor} \) (P9)
- \( P_{10} = \text{has\_attr Dave role patient} \) (P10)
- \( P_{11} = \text{has\_reln Alice Bob doctor\_of} \) (P11)

The formula \( s_2 \) inferred by infer by using \( x = \{ k, k', k'', p \} \) and substituting \( k \) for Alice, \( k' \) for Charlie, \( k'' \) for Dave and \( p \) for Dave\_PHI is:

\[
\forall k, k', k'' : \text{principal}. \forall p : \phi. \\
\text{has\_attr } k \text{ role doctor } \land \text{has\_attr } k' \text{ role doctor } \land \text{owner p } k'' \land \text{has\_reln } k k'' \text{ doctor\_of } \land \text{has\_attr } k'' \text{ role patient} \\
\square \text{may send } k p k' \text{ (purpose:treatment)}
\]

(F2)

The function reduce takes in the formula above \((s_2)\) and the one generated before, \( s_1 \) \( \{\text{F1}\} \), and reduces it to the formula above \((s_2)\) as it is less strict. The output of reduce is True. While the reduction may lead to incorrect formulas being added as part of the inferred policy, it can also remove unnecessary predicates from the formula. For instance, suppose Alice is also a director in the healthcare organization at times \( T \) and \( T' \). An additional predicate entailed from the database could be

\( P_{12} = \text{has\_attr Alice role director} \) (P12)

which is added to the formulas \( s_1 \) and \( s_2 \). However, other instances of a doctor (who does not have any other role) sending a patient’s information to another doctor might not include \( P_{12} \) generating a more precise formula. With reduce the predicate \( P_{12} \) will be correctly masked in the final inferred policy. This also emphasizes the need for an iterative process over enormous logs to generate meaningful formulas with necessary predicates.

3) Generate: Algorithm 5 lists our policy inference algorithm, Generate. Given a list of log entries \( L \), an existing policy \( (\varphi_0) \), a typing environment \( (\Sigma) \) and a set of relations \( (R) \), Generate starts by inferring a formula for every log-entry in \( L \) using infer. It then reduces the set of formulas using the function reduce and returns a map \( M \) containing the mapping from the reduced formulas to the set of original formulas. The abstract function appendMap appends (updates) the key \( s_1 \) in the map with the value \( s_2 \) and returns the new map. The map \( M \) is useful when a violation is detected by the oracle in the inferred policy and helps recover the original formulas in case the reduced version isn’t a valid formula in the policy.

**Algorithm 5:** Policy Inference Algorithm

**Input:** \( L \) - a list of log entries  
\( \varphi_0 \) - existing policy  
\( \Sigma \) - typing environment  
\( R \) - relational database

**Output:** \( \varphi_n \) - inferred policy

\( M \) - map from formula to set of formulas

**Function Generate** \( \langle L, \varphi_0, \Sigma, R \rangle \):

1. \( \varphi_o \leftarrow \text{infer}(L, R, \Sigma); \varphi_n \leftarrow \varphi_0; M \leftarrow []; \)
2. foreach \( s_i \in \varphi_0 \) do
3. foreach \( s_j \in \varphi_o \setminus \{ s_i \} \) do
4. if reduce\((s_i, s_j, \Sigma)\) then
5. \( \varphi_o \leftarrow \varphi_o \setminus s_j; \)
6. \( M \leftarrow \text{appendMap}(M, s_i, s_j); \)
7. else if reduce\((s_j, s_i, \Sigma)\) then
8. \( \varphi_n \leftarrow \varphi_n \setminus s_j; \)
9. \( M \leftarrow \text{appendMap}(M, s_j, s_i); \)
10. break;
11. end
12. end
13. end

**End Function**

**B. Properties of Inference Algorithm**

Our algorithm satisfies a few important properties:

- **Monotonicity:** For a fixed set of relations, adding more entries to a log with an inferred policy \( \varphi \) results in a more or equally permissive policy.
- **Termination:** Each iteration of the algorithm terminates.
- **Soundness:** The log from which our algorithm infers a policy satisfies the inferred policy.
- **Minimality:** The policy inferred by our algorithm is the most restrictive policy (that can be inferred by our algorithm) that the log satisfies.

We discuss each of these properties next and prove them for our algorithm.

1) **Monotonicity:** As our inference algorithm only adds policies to the existing set of policies by gathering more information over time, it satisfies the property of monotonicity. We show that the policy inferred by the inference algorithm (Algorithm 5) becomes more permissive as the algorithm parses more log entries. To prove this property, we establish a few important results.

**Lemma 3.** Given a typing environment \( \Sigma \), a set of relations \( R \), a log \( L \), and formulas \( s \) and \( s' \), if \( L \) satisfies either the formula \( s \) or \( s' \), then it satisfies the policy containing both \( s \) and \( s' \).

**Lemma 4.** Given a typing environment \( \Sigma \), a set of relations \( R \), a log \( L \), and formulas \( s \) and \( s' \), if \( L \) satisfies the formula \( s \), and reduce\((s', s, \Sigma)\) returns True, then \( L \) satisfies the formula \( s' \).

**Lemma 5.** Given a typing environment \( \Sigma \), a set of relations \( R \), and two log entries \( \gamma \) and \( \gamma' \), if our algorithm infers the
policy $\phi_1$ for $\gamma$ and $\phi_2$ for $\gamma', \gamma''$, then $\phi_1 \leq \phi_2$.

**Theorem 1.** Given a typing environment $\Sigma$ and a set of relations $R$, if our algorithm infers a policy $\phi_1$ for the log $\mathcal{L}$ and infers a policy $\phi_2$ for the log $\gamma, \mathcal{L}$, then $\phi_1 \leq \phi_2$

As a consequence of the monotonicity theorem, we can show that as new logs are generated the inference will eventually reach a fixed-point because the number of formulas in the system are bounded and finite as mentioned earlier. To show that the inferred policy eventually reaches a fixed-point where no more formulas can be added to it, we use Tarski’s fixed-point theorem [32]. For this, we need to ensure that our inference algorithm, Generate, that returns a new policy using the logs is monotonically increasing, which follows from Theorem 1.

2) **Termination:** The logs might not represent all possible transactions or requests, and hence, the actual policy enforced in the system. The inference is iterative and adds formulas as and how log entries are parsed by the algorithm. Given that the entities in the system we consider are finite and bounded, our algorithm is terminating. The proposition that the algorithm terminates follows trivially from the fact that the number of log entries input to the algorithm are finite. This can further be optimized to exclude the existing set of formulas in $\varphi_a$, and check for only the current set of inferred formulas.

3) **Soundness:** The policy generated by our algorithm from the log entries should suffice to “accept” those log entries as valid. We show that the policy inferred by our algorithm is meaningful by proving that our algorithm is sound. More specifically, we show in Theorem 2 that the policy generated by our algorithm accepts all log entries from which it was generated. In other words, the log $\mathcal{L}$ satisfies the policy generated from $\mathcal{L}$.

**Theorem 2.** Given a log $\mathcal{L}$, a typing environment $\Sigma$ and a set of relations $R$, if our algorithm generates a policy $\varphi$, then the log $\mathcal{L}$ satisfies $\varphi$.

4) **Minimality:** A trivial policy that accepts all possible log entries can prove that the algorithm is sound but is overly-permissive. An important property that we prove about our algorithm is that the policy inferred is minimal, i.e., the policy generated by our algorithm is the most restrictive policy that can process all log entries from which it was generated.

**Lemma 6.** Given a log $\mathcal{L}$, a typing environment $\Sigma$ and a set of relations $R$, and any policy $\varphi'$, if $\forall \gamma \in \mathcal{L}. \text{Generate}(\gamma, \{}\}, \Sigma, R) = (\{}\}, \Sigma)$ and $\Sigma; R; \gamma \vdash \varphi'$, then $\exists s' \in \varphi'. (s = s' \vee \text{reduce}(s', s, \Sigma) = \text{True})$

**Proof.** Let $\gamma = r(\bar{t}) \in \mathcal{L}$. From Theorem 6 we have $\Sigma; R; \gamma \vdash \{s\}$ such that $s$ contains all ownerships, attributes and relationships related to $\bar{t}$. From Lemma 6 $\exists s' \in \varphi'. (s = s' \vee \text{reduce}(s', s, \Sigma) = \text{True})$.

We now show that the policy generated by our algorithm is sound. To do this, we need to prove that the policy generated by our algorithm is sound. To show that the policy is sound, we need to prove that the policy generated by our algorithm is sound. To show that the policy is sound, we need to prove that the policy generated by our algorithm is sound. To show that the policy is sound, we need to prove that the policy generated by our algorithm is sound. To show that the policy is sound, we need to prove that the policy generated by our algorithm is sound.

**Algorithm 6:** Definition of oracle($\varphi, M$)

**Input:** $\varphi$ - policy
$M$ - map of formulas to list of formulas

**Output:** $\varphi_a$ - approved policy

**Function oracle ($\varphi, M$):

1. $\varphi_a \leftarrow \{\}; \varphi_m \leftarrow \{\}$
2. foreach $s \in \varphi$
doi {isValid(s) then
3. $\varphi_a \leftarrow \{s\} \cup \varphi_a$
4. else
5. $\varphi_a \leftarrow M(s) \cup \varphi_a$
6. end
7. end
8. foreach $s \in \varphi_a$
do {isValid(s) then
9. $\varphi_a \leftarrow \{s\} \cup \varphi_a$
10. end
11. end
End Function

**Theorem 3.** Given a log $\mathcal{L}$, a typing environment $\Sigma$ and a set of relations $R$, if Generate($\mathcal{L}, \{}\}, \Sigma, R) = (\varphi, M)$, then $\forall \varphi', \Sigma; R; \mathcal{L} \vdash \varphi' \Rightarrow \varphi \leq \varphi'$

**Proof.** From assumption, we have $\forall \varphi'. \Sigma; R; \mathcal{L} \vdash \varphi'$. From Theorem 2, we have $\Sigma; R; \mathcal{L} \vdash \varphi$. To show: $\forall \Sigma'; R'; \mathcal{L}'; \Sigma, R; \mathcal{L} \vdash \varphi \Rightarrow \Sigma'; R'; \mathcal{L}' \vdash \varphi'$. As $\Sigma'; R'; \mathcal{L}' \vdash \varphi$, from Lemma 2, $\forall \gamma \in \mathcal{L}', \exists s \in \varphi. \Sigma; R'; \gamma \vdash \{s\}$. From Lemma 6, we have a corresponding $s' \in \varphi'$. Hence, the conclusion holds.

**C. Oracle and Violation Detector**

An important application of our framework is automated auditing of logs and detecting violations. Our inference algorithm generates a set of formulas that are audited by an oracle. The entries corresponding to the formulas that are accepted by the oracle are allowed while the violations as per the approved policy are marked for further review. We describe the functionality of the oracle and how the framework detects violations in more detail below.

1) **Oracle:** Algorithm 6 defines the functionality of the oracle. The oracle takes in the generated policy $\varphi$ along with the map $M$ and returns the approved policy $\varphi_a$. For an empty set of formulas, the oracle returns an empty set. If a formula $s$ is a valid formula in the policy, the oracle returns it as part of the approved policy (line 12). The oracle uses an abstract function isValid (related to the type of oracle), to determine whether or not $s$ is an acceptable formula in the policy (line 5). If the oracle detects an invalid formula $s'$ according to the policy, it does not return as part of the validated policy. However, as the reduce function attempts to minimize the number of formulas that the oracle should audit, it might mask a violation by producing a relaxed policy that contained the invalid formula $s'$ instead of the more strict formula(s). To account for this reduction, the oracle uses
map $M$ generated by Generate to obtain the set of formulas that $s'$ maps to (line 6) and validates them instead (line 9).

a) Example: We show the functionality of oracle in deciding the approved policy using the example scenario described before. Suppose that $s_1$ (Fig. 1) is a valid formula in the policy but $s_2$ (Fig. 2) isn’t because a doctor requires authorization by the patient to send the patient’s PHI to another doctor. The inputs to the oracle are the policy and the map: $\{s_2\}, [\{s_2 \mapsto \{s_1\}\}]$. As $s_2$ is not a valid formula in the policy, isValid($s_2$) returns False. Then, the map is looked up to retrieve the set of formulas that $s_2$ maps to, which is $\{s_1\}$. The updated set of formulas that the oracle has to check is now $\{s_1\}, [\{s_2 \mapsto \{s_1\}\}]$. As $s_1$ is an acceptable formula in the policy, the oracle adds it to the validated policy and returns $\{s_1\}$ as the validated policy.

2) Violation Detection: Once the oracle has validated the policy, the logs are run against the validated policy to detect any violations. The validation of logs against the policy is done using the rules described in Figure 6. For every log entry $\gamma = \langle A, U, o, einfo@T\rangle$, the enforcement checks if $\gamma$ satisfies the approved policy $\varphi_a$. If $\Sigma; R; \exists A, U, o, einfo@T \models \varphi_a$, then the access is an authorized access; if not, the entry is marked for further review.

a) Example: We illustrate how violations are detected for the above example. The enforcement starts by checking the policy for transaction $T_1$ as shown in Figure 5. The enforcement starts by using the INST rule by replacing $\bar{x}$ in $s_1$ with terms $\bar{t}$. It then applies the PRED rule to check the predicates in $s_1$ with the substitution. Using rules ATTR, OWN and RELN, the relevant records are found in the database $R$, which completes the validation of transaction $T_1$. The derivation uses $P_i$ that correspond to the predicates $P_1, P_2$ to $P_{11}$ defined above in the example for $s_1$. Figure 6 attempts to validate transaction $T_2$. It proceeds in a similar fashion as transaction $T_1$ using the predicates $P_7$ to $P_{11}$ but does not find a relevant entry for establishing $\exists h_{\text{reln}}\text{Dave doctor_of}$ in $R$ thereby reporting the transaction as a violation ($f$).

D. Discussion

The policy generated is essentially an access control policy that defines who is allowed access to what type of data and when. In particular, the policy is attribute- and relationship-based and uses environmental state on the side. While the roles do not have a hierarchical structure in our representation, the organization might have this structure in place. This makes reasoning about the policy simpler as we do not have to consider the role-hierarchy when making decisions. The hierarchical order and delegations are captured through relationships between the different users in the system, which have a specific timeframe similar to granting and revoking access in a role-based access control system.

V. Implementation

We have implemented the algorithms and functions described above in Python. We describe their implementation and related optimizations next.

The function infer loops on the number of log entries and generates policies. It takes in $R$ and $\Sigma$ as arguments. In the implementation, the log is represented as a table in the database $R$. The relationships and associations in $R$ are implemented as separate tables with time-related information. The resources and principals are stored separately with attribute information, and the environment $\Sigma$ contains mapping from the resources and principals to either the table name or table-column name pair. We separately store the list of attributes to consider for different types of resources and principals, and ensure that they have a finite domain. Instead of visiting every record in the database related to relationships and associations, we do a standard lookup using a SQL query to return the results.

Our implementation assumes that only one resource is accessed every transaction and can involve at most three principals (one being the owner of the resource). Thus, the formulas generated deal with at most three principals, their (fixed) attributes, and the relationships between them. As all of this is pre-determined, the policy has a fixed structure and is represented as a table in our implementation. The rows of the
table are the formulas that form the policy. Fields that cannot be populated for any of the formulas are set to NULL.

Once the formulas are generated by the algorithm, the oracle needs to decide on the validity of the formulas as part of the policy. The oracle is an important component in our model as it decides the legitimacy of the policy inferred by our algorithm. The oracle could be any function depending on the log under consideration, the application, and other heuristics. In our implementation, we manually check the formulas to determine the correct policy. While we employ a human oracle, different types of oracles can be considered to determine validity.

For instance, the simplest oracle function is the identity function that returns all policies inferred by the algorithm. This oracle is useful when working with pre-audited logs and can be used when initially bootstrapping the system with policies. The organization can use the logs that have been audited earlier to generate meaningful policies that can be enforced and checked against when inferring newer policies. Existing works employ machine-learning algorithms to generate policies from logs and access-requests [21–31]. A more complex oracle may employ a similar machine-learning algorithm to determine the correctness and validity of the inferred policy based on some threshold. Determining this threshold is, however, tricky, and might result in false-positives or over-permissive formulas.

VI. Evaluation

Our evaluation of the algorithm aims to answer three important questions — (1) is auditing policy easier than auditing logs, i.e., are the number of formulas lesser than the number of log entries and how does this ratio vary for different number of log entries; (2) is the violation detector able to detect violations of access policy in the log using the formulas that fail the policy audit; (3) how permissive and expressive are the formulas generated by our algorithm as compared to the actual policy. We evaluate our implementation for these questions by using it to generate policies and detect violations in the case of healthcare systems, which we discuss next.

A. System Setup

The information that is sensitive in a healthcare setting is the protected health information (PHI) of a patient that includes psychotherapy notes amongst other information. The HIPAA privacy rule places restriction on how this information can be accessed and/or disclosed to a third-party by a user. Thus, the audit logs in such systems might contain additional information about a receiver to whom the disclosure is made. The HIPAA privacy rule defines various clauses in which it is appropriate for a covered entity to access a patient’s PHI, which are formalized as purposes in prior work [18]. These may, however, be very subjective in nature and require consideration on a case-by-case basis. Our enforcement includes purpose as part of the policy and records it as a separate column as is consistent with policy representation in hippocratic databases, which are a class of databases that assist in reasoning about the privacy of healthcare information they manage. We start with an initially empty set of formulas.

We test our algorithm on synthetic logs generated by a simulation that considers disclosure scenarios governed by HIPAA. We modify the implementation by Garg et al. [14] written in C to simulate the log generation process; the original implementation generated predicates that were used by their audit algorithm directly. The simulation generates data disclosing log entries for different purposes like treatment, billing, health-operations, law-enforcement and marketing. It additionally generates attributes like roles, and relationships for users and resources. The simulated database is an instance of hippocratic database. The simulator uses a probabilistic event scheduler for generating data disclosure for each purpose after a probabilistic gap. It also generates violations with a probability input to the simulator by omitting generation of certain necessary conditions for a log entry to be valid. The log is a table in the database containing disclosure transactions similar to the example from Section IV. When generating the attributes and relationships, it adds them to a separate table to verify that the relationships and attributes have a finite domain of values.

The reduce function reduces the number of formulas that the oracle has to initially review by mapping the less strict formulas to the more strict formulas, and only adding the mapped formulas if the less strict formulas are invalidated by the oracle. In our implementation less strict formulas are those which do not have a value specified for some of the fields, e.g., while one formula has a relationship specified, the less strict formula may not have a relationship. Thus, if infer infers the policy $\\{s_1, s_2, s_3\}$ such that $s_1 = (a_1 \land a_2) \supset \text{may } r$, $s_2 = (a_1 \land a_2 \land a_3) \supset \text{may } r$ and $s_3 = (a_1 \land a_2 \land a_4) \supset \text{may } r$, then the policy is reduced to $\{s_1\}$ with a mapping: $s_1 \mapsto \{s_2, s_3\}$.

B. Performance Evaluation

We perform two experiments — one without any violations, and the other having violating instances of HIPAA with a probability of 0.1. All experiments were performed on a 3.1 GHz Dual-Core Intel Core i5 CPU running macOS Catalina with an 8GB RAM. The database we work with is SQLite version 3.31.

In the first experiment, to address the first question, we evaluate the time taken by our algorithm to infer policy for different number of log entries that are increased additively, i.e., the logs contains all entries from previous experiment apart from the new ones that are added by the log generator. An important result of this experiment is to show the saturation of formulas generated over time as new log entries are added to the log. Table I summarizes the results. The first column contains the number of log entries input to the algorithm. The second column lists the number of formulas inferred for the given log and the third column shows the ratio of the number of formulas to the number of log entries in percentage. The fourth column reports the time taken to perform the inference in seconds while the fifth column shows the average time taken to process one log entry in milliseconds. Figure 7 shows the
Number of inferred formulas quickly stabilize once 50,000 log entries are processed. The majority of time taken for inferring formulas is spent when newer formulas are inferred that do not match any of the earlier formulas. We create indexes on the policy table for faster lookup once all the policies are inferred. Thus, for every 50,000 entries past 100,000 entries, the inference takes on an average 62.5 seconds. The formulas contain a representative set of clauses used on a more frequent basis. It is also informative to see the ratio of formulas required to be audited to the actual number of log entries. For 12,500 log entries, the oracle has to audit only 287 formulas, which is only 2.29% of the number of log entries. The percentage goes down significantly to 0.112% for 350,000 entries supporting the use of inference for auditing.

We apply reduce to the formulas generated by the algorithm and find that the number of formulas goes down by 143 for 350,000 log entries, thus further reducing the number of formulas that the oracle has to approve.

To answer the second question in our evaluation, in the second experiment we measure the effectiveness of violation detection by the algorithm and compare the number of violating log entries with the number of formulas generated by the algorithm that are not part of the policy. We simulated the generation of logs to model policy violations with a probability of 0.1. The simulator generated 300,000 log entries. Most violations included missing authorizations to access a patient’s personal health information. The number of formulas inferred by our algorithm is 206. While the reduce function further reduced the number of formulas by 33, we audited all 206 formulas to ensure no violations were missed. Of that, 8 formulas were incorrect as per the HIPAA regulations, and 2 were part of the reduced formulas. For instance, one of the formulas allowed a patient’s protected health information to be released for marketing purposes without the consent of or authorization from the patient. Similarly, another formula allowed the release of PHI for treatment purposes to another covered entity without the consent of the patient violating §164.508(a)(2) of the HIPAA rule. In total, the 8 incorrect formulas corresponded to 28,796 log entries. To summarize the second experiment, auditing 206 formulas helped uncover around 29k violating log entries.

C. Qualitative Evaluation

To address the third question, and understand the permissiveness and expressiveness of our algorithm, we compare the formulas inferred by our algorithm with the actual HIPAA policy expressed in first-order logic [18]. The inferred formulas are specific to the log entries, and hence, might be more restrictive than the actual HIPAA policy. For instance, one of the formulas inferred in our experiments is:

\[ \forall k, k', k'': \text{principal} \forall p : \text{phi}. \]

\[ \text{has_attr} \ k \ \text{role c-entity} \land \text{has_attr} \ k' \ \text{role c-entity} \land \]

\[ \text{owner} \ p \ k'' \land \text{has_reln} \ k \ k'' \ \text{hospital_of} \land \]

\[ \text{has_reln} \ k' \ k'' \ \text{health_care_provider_of} \land \]

\[ \text{has_attr} \ k'' \ \text{role patient} \]

\[ \rightarrow \text{may send} \ p \ k' \ (\text{purpose:payment}) \]

The actual HIPAA policy (§164.506(c)(3)) does not require \(k''\) to be both a covered-entity and the health-care-provider of \(k''\), which makes the formula inferred by our algorithm stricter.

Suppose if for another log-entry, the formula inferred by our algorithm is,

\[ \forall k, k', k'' : \text{principal} \forall p : \text{phi}. \]

\[ \text{has_attr} \ k \ \text{role c-entity} \land \text{has_attr} \ k' \ \text{role c-entity} \land \]

\[ \text{owner} \ p \ k'' \land \text{has_reln} \ k \ k'' \ \text{hospital_of} \land \]

\[ \text{has_reln} \ k' \ k'' \ \text{role patient} \]

\[ \rightarrow \text{may send} \ p \ k' \ (\text{purpose:payment}) \]

then the earlier entry is moved to the map by reduce, and the less strict formula is presented to the oracle. Thus, as more log entries are parsed by our algorithm, the policy converges towards the actual policy enforced in the system.

Our algorithm infers policy clauses with disjunction operator as separate formulas that are stricter than or equivalent to the actual policy. For instance, our implementation infers the following three individual formulas,

\[ \forall k, k', k'' : \text{principal} \forall p : \text{phi}. \]

\[ \text{has_attr} \ k \ \text{role c-entity} \land \text{has_attr} \ k' \ \text{role c-entity} \land \]

\[ \text{owner} \ p \ k'' \land \text{has_reln} \ k \ k'' \ \text{hospital_of} \land \]

\[ \text{has_attr} \ k'' \ \text{role patient} \]

\[ \rightarrow \text{may send} \ p \ k' \ (\text{purpose:payment}) \] \hspace{1cm} (1)

\[ \ldots \rightarrow \text{may send} \ p \ k' \ (\text{purpose:treatment}) \] \hspace{1cm} (2)

\[ \ldots \rightarrow \text{may send} \ p \ k' \ (\text{purpose:healthcare-operations}) \] \hspace{1cm} (3)
DeYoung et al. [18] formalize the clause § used in practice as more log entries are parsed. For instance, “some” value. The algorithm generates more possible values this is more restrictive than the original clause which requires an or what the existential formula is that a log-entry satisfies; with specific values as it is difficult to anticipate whether it is

The inferred policy might not be the actual policy but an

Our inference instantiates r1 and r2 as hospital-of when inferring this formula.

B. Temporal formulas

Disclosures like “... the covered entity must act on a request for access no later than 30 days after receipt of the request ...” cannot be inferred by our algorithm. Such clauses pertain to timely disclosure of data, and not the actual access of data. While the current algorithm does not generate temporal formulas and formulas like “... can access only three times ...”, it can be extended to do so using the timing information (and frequency of access) in the log and database entries, which is an interesting future direction and out of the scope of the current work. The logs have to be treated by an additional processing step to record the frequency of access and the time between accesses. The algorithm would then take into consideration the frequency and time of access of data, along with other accesses to formulate the respective policy.

VIII. RELATED WORK

We discuss three classes of research that are closely related to our work: auditing logs based on policy specifications, mining access control policies, and mining properties from logs. While a lot of work has focussed on these three classes separately, we are not aware of any work that infers access policies from logs for auditing them.

A. Auditing Logs

Auditing logs based on policy specifications has been a topic of active research over the last decade [12–17]. Instead of employing a runtime monitor to check the policies, some of these techniques have focussed on monitoring the logs generated to check for policy violation while others model and enforce the accountability requirements in systems [34, 35]. All of these systems including ours assume correct logging practices; however, some prior work also discusses and formalizes correct logging [36]. We discuss the works on auditing logs in more detail next.

### Table I: Experimental evaluation of inference algorithm to generate policy for different number of log entries

<table>
<thead>
<tr>
<th># log entries</th>
<th># formulas inferred</th>
<th>% entries</th>
<th>time taken (sec)</th>
<th>average time per entry (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12500</td>
<td>287</td>
<td>2.296</td>
<td>105</td>
<td>8.75</td>
</tr>
<tr>
<td>25000</td>
<td>352</td>
<td>1.408</td>
<td>227</td>
<td>9.08</td>
</tr>
<tr>
<td>50000</td>
<td>391</td>
<td>0.782</td>
<td>441</td>
<td>8.82</td>
</tr>
<tr>
<td>100000</td>
<td>391</td>
<td>0.391</td>
<td>602</td>
<td>6.02</td>
</tr>
<tr>
<td>150000</td>
<td>391</td>
<td>0.261</td>
<td>670</td>
<td>4.46</td>
</tr>
<tr>
<td>200000</td>
<td>391</td>
<td>0.195</td>
<td>731</td>
<td>3.65</td>
</tr>
<tr>
<td>250000</td>
<td>391</td>
<td>0.156</td>
<td>780</td>
<td>3.12</td>
</tr>
<tr>
<td>300000</td>
<td>391</td>
<td>0.130</td>
<td>851</td>
<td>2.84</td>
</tr>
<tr>
<td>350000</td>
<td>391</td>
<td>0.112</td>
<td>914</td>
<td>2.61</td>
</tr>
</tbody>
</table>

while they are represented as a single clause with disjunctions in the formalized HIPAA policy §164.506(c)(1) [18]. Note that the individual formulas are inferred only when the relevant log entries are parsed by the algorithm.

VIII. RELATED WORK

We discuss three classes of research that are closely related to our work: auditing logs based on policy specifications, mining access control policies, and mining properties from logs. While a lot of work has focussed on these three classes separately, we are not aware of any work that infers access policies from logs for auditing them.

A. Auditing Logs

Auditing logs based on policy specifications has been a topic of active research over the last decade [12–17]. Instead of employing a runtime monitor to check the policies, some of these techniques have focussed on monitoring the logs generated to check for policy violation while others model and enforce the accountability requirements in systems [34, 35]. All of these systems including ours assume correct logging practices; however, some prior work also discusses and formalizes correct logging [36]. We discuss the works on auditing logs in more detail next.
Cederquist et al. [12] present a system for auditing by discharging formal obligations through construction of formal proofs. The process checks that an obligation is satisfied somewhere in the proof-tree, the leaves of which are established using the logs. Our work does not require a formal representation of the policy, and hence a specification to work with for auditing.

Garg et al. [14] work with incomplete logs by reducing the policy to validate based on available logs. As more entries are made available, their system reduces the policy further until either the policy has a definitive truth value or contains only subjective clauses. The policies in their work are specified in an expressive first-order logic. A similar work by Basin et al. [15] targets compliance checking in incomplete logs by specifying policies in a first-order logic, which is a variant of the logic used by their previous work [19] for runtime monitoring of policies. They have a stronger approach that is suitable for online monitoring. Our enforcement algorithm is formalized in a similar first-order logic but does not include subjective clauses while our approach is not dependent on policy specification.

APPLE [13] is a logical framework that uses logs to determine violations of policies in a system. The system ensures that only those users who can be held accountable in the system can access resources. Their focus is to detect violations and hold users accountable for accessing resources that have policies associated with them. In contrast, our work targets systems where formal policy specifications are unavailable, and assists in auditing the logs by inferring policies.

The Aura programming model [16], on the other hand, uses a set of rules that constitute a policy, and defines a kernel containing the log and the resource with an interface that interacts on behalf of the resource. The interface takes as input an authorization proof that validates the access. The kernel returns a corresponding proof to indicate that the operation was performed. The logs serve as evidence of access, which is similar to what we base our analysis on. However, they do not infer the policy, and instead base the compliance on recorded proofs.

Fabbri and LeFevre [37] study the problem of generating explanations for individual records in the log. Their work generates explanations for every entry based on some template to justify the log-entry. Our approach parses logs in a similar fashion but generates concrete policies that can be applied at runtime. More recently, Chowdhury et al. [38] proposed a hybrid approach to perform both online monitoring to identify violations and an offline auditing approach for cases it cannot validate. Our approach is similar to theirs in that we can perform runtime access control based on the earlier inferred policies while for the transactions that do not have a corresponding policy, we log them for further examination.

B. Mining access control policies

Mining policies has been a well-studied problem [20–31]; almost all approaches for mining policies are based on using learning algorithms and generalizing/correcting the results to obtain the policy enforced by the system. We do not base our inference on a machine learning technique to avoid pitfalls of incorrect authorizations. Instead, we use rules from authorization logic to decide the access permission of different users in the system. Some other works have focussed on role-mining and role-engineering [39–42], which aims to mine user-roles (and hierarchies) for designing and enforcing role-based access control in systems. We discuss some of the approaches to policy mining next.

Agarwal et al. [43] proposed the Apriori algorithm for mining association rules between various items in a database. Most of the subsequent works on policy mining have used variations of this algorithm to mine meaningful policies from the transactions. Bauer et al. [21] proposed a methodology to apply this algorithm to mine attribute-based access control policies from access logs of a lab to determine the access patterns of different users. The patterns helped them detect any misconfigurations in the enforced policy, similar to how we use an oracle to detect violations of policies.

More recently, Cotrini et al. [29] presented Rhapsody that uses association rule mining to mine logs for discovering attribute-based access policies. The mined policies have a minimal over-privilege, and is designed to work with sparse logs, i.e., if not all kinds of transactions are present in the logs. In contrast, we do not generalize for transactions that are not represented in the logs and wait until such a transaction actually appears to disallow any unpredicted authorizations.

Sanders and Yue [31] present a rule mining algorithm using privilege error minimization [44] to generate least privilege policies. Their focus was to measure the under-privilege and over-privilege of policies and to minimize them in large and complex systems.

Xu and Stoller [22–25] proposed different approaches to mine role-based and attribute-based access control policies. They proposed various algorithms to efficiently mine these policies while minimizing the size of the policies mined. While their initial work [23] on mining attribute-based policies used the role matrix, their subsequent work [25] mined such policies from logs, similar to what a part of our system does. They, however, use machine learning techniques, and weighted structural complexity [40] as a measure for the size of policies they generate. Bui et al. [27, 30] have used modifications of these algorithms to mine relationship-based access control policies. But their approaches use access-control lists and attribute data to determine these policies. Our approach, on the other hand, infers both relationship-based and attribute-based access policies using logs and attribute data.

While our approach and the previously proposed approaches focus on generating authorization policies that allow access, Iyer and Masounzadeh [28] present an algorithm for mining both positive and negative policies, i.e., policies that disallow access. They also use weighted structural complexity to measure the complexity of the policies and show that their mining algorithm has a better performance than Xu and Stoller’s algorithm [24].
C. Mining properties from logs

A lot of other prior work focusses on inferring properties from logs but differ on what properties their system infers from the input logs. We discuss briefly the approaches that mine properties other than access policies from logs here. Most of these approaches analyze traces generated by the system either to understand the behavior of the program by mining specifications for existing systems or to visualize how the user interacts with the system, and to determine the faults and issues with the system. Ammons et al. present one of the earliest approaches to mining program specifications from execution traces using machine learning algorithms. IPM2 performed interaction-pattern mining from logs using machine learning.

Most of these approaches analyze traces generated by the system to mine properties other than access policies from logs here. The faults and issues with the system. Ammons et al. present one of the earliest approaches to mining program specifications from execution traces using machine learning algorithms. IPM2 performed interaction-pattern mining from logs using machine learning.

We present our inference algorithm and evaluate our implementation for a simulated set of healthcare audit logs. We show that our inference algorithm generates significantly lesser formulas than log entries making them easier to audit, and can effectively identify violations in logs.

IX. CONCLUSION

We have presented a novel methodology to automate auditing of logs by policy inference. Our algorithm infers formulas from log entries, which are then validated by an oracle that can be used to automatically audit logs. We prove that our algorithm is sound, terminates, and generates a minimal policy. We implement our algorithm and evaluate our implementation for a simulated set of healthcare audit logs. We show that our inference algorithm generates significantly lesser formulas than log entries making them easier to audit, and can effectively identify violations in logs.

ACKNOWLEDGMENTS

We would like to thank our anonymous shepherd, and the anonymous reviewers for their insightful comments and feedback. We would also like to thank Deepak Garg and Limin Jia for initial discussions on the formalization and implementation. The work was supported in part by the Center for Machine Learning and Health (award no. PO0006063506) at Carnegie Mellon University.

REFERENCES


Appendix

A. Lemmas and Proofs

Lemma 1. For all policies \( \varphi_1 \) and \( \varphi_2 \), and a formula \( s \), if \( \varphi_1 \leq \varphi_2 \), then \( \varphi_1 \cup \{ s \} \leq \varphi_2 \cup \{ s \} \).

Proof. As \( \varphi_1 \leq \varphi_2 \), we have

(1) \( \forall \Sigma, R, \mathcal{L}; \Sigma; R; \mathcal{L} \models \varphi_1 \implies \Sigma; R; \mathcal{L} \models \varphi_2 \).

Suppose that \( \forall \Sigma', R', \mathcal{L}', \Sigma'; R'; \mathcal{L}' \models \varphi_1 \cup \{ s \} \).

T.S.: \( \Sigma'; R'; \mathcal{L}' \models \varphi_2 \cup \{ s \} \).

Proof follows by induction on the length of the log \( \mathcal{L}' \) and then applying ACCESS along with (1).

Lemma 2. Given a policy \( \varphi \), for all logs \( \mathcal{L} \), typing environments \( \Sigma \) and sets of relations \( R \), if \( \Sigma; R; \mathcal{L} \models \varphi \), then \( \forall \gamma \in \mathcal{L}, \exists s \in \varphi. \Sigma; R; \gamma \models \{ s \} \).

Proof. Proof follows by induction on the length of the log \( \mathcal{L} \). Base case: \( \mathcal{L} = \{ \gamma \} \).

From IND, \( \Sigma; R; \gamma \models \varphi \). From ACCESS, we have \( \Sigma; R; \gamma \models \{ s \} \) such that \( s \in \varphi \). Hence, the conclusion holds.

Ind. case: \( \mathcal{L}' = \gamma', \mathcal{L} \).
IH: If $\Sigma; R; L \models \varphi$, then $\forall \gamma \in L, \exists s \in \varphi; \Sigma; R; \gamma \models \{s\}$
From IND, $\Sigma; R; \gamma \models \varphi$ and $\Sigma; R; L \models \varphi$. From IH, we have
$\forall \gamma \in L, \exists s \in \varphi; \Sigma; R; \gamma \models \{s\}$
T.S. $\forall \gamma \in [\gamma], \exists s \in \varphi; \Sigma; R; \gamma \models \{s\}$
From IND, $\Sigma; R; \gamma' \models \varphi$. From ACCESS, we have $\Sigma; R; \gamma' \models \{s\}$ such that $s \in \varphi$. Hence, the conclusion holds.

**Lemma 3.** Given a $\Sigma, R, L$, and formulas $s$ and $s'$, if $\Sigma; R; L \models \{s\}$, then $\Sigma; R; L \models \{s, s'\}$

**Proof.** By induction on the length of log $L$.
- Base case: $|L| = 1$. From ACCESS, and $\Sigma; R; L \models \{s\}$, we have $\Sigma; R; L \models \{s, s'\}$.
- Ind. case: IH: If $\Sigma; R; L \models \{s\}$, then $\Sigma; R; L \models \{s, s'\}$
  
  Thus, from IND, the conclusion holds.

**Lemma 4.** Given a $\Sigma, R, L$, and formulas $s$ and $s'$, if $\Sigma; R; L \models \{s\}$ and reduce($s', s, \Sigma$) = True, then $\Sigma; R; L \models \{s'\}$

**Proof.** By induction on length of log $L$.
- Base case: $|L| = 1$. Let $r(i) @ T = L$, $s = (\bigwedge_{i=1}^{m} a_i(x_i)) \supset \bigwedge_{i=1}^{m} a_i(x_i) \supset \bigwedge_{i=1}^{m} a_i(x_i) = m \supset m \supset m$. From Algorithm [4], we know that $r(i) = r'(i')$ and $\forall i \in \{1, \ldots, m\}, a_i(x_i) \in \{a_i(x_i)\}_{i=1}^{m}$ such that $m \leq n$.
  
  From PRED and $\Sigma; R; r(i) @ T \models \{s\}$, we know $\Sigma; R; T \vdash \{a_i(x_i)\}_{i=1}^{m} \supset \{a_i(x_i)\}_{i=1}^{m}$. As $\forall i \in \{1, \ldots, m\}, a_i(x_i) \in \{a_i(x_i)\}_{i=1}^{m}$, from PRED, we have $\Sigma; R; r(i) @ T \models \{s'\}$. Hence, the conclusion holds.
- Ind. case: IH: If $\Sigma; R; L \models \{s\}$ and reduce($s', s, \Sigma$) = True, then $\Sigma; R; L \models \{s'\}$
  
  T.S. If $\Sigma; R; \gamma, L \models \{s\}$ and reduce($s', s, \Sigma$) = True, then $\Sigma; R; \gamma, L \models \{s, s'\}$
  
  From IND, $\Sigma; R; \gamma \models \{s\}$ and $\Sigma; R; L \models \{s\}$
  
  From similar reasoning as in base case, we know that if $\Sigma; R; \gamma \models \{s\}$, then $\Sigma; R; \gamma \models \{s'\}$
  
  From IH, if $\Sigma; R; L \models \{s\}$, then $\Sigma; R; L \models \{s'\}$
  
  Thus, from IND, the conclusion holds.

**Lemma 5.** Given a $\Sigma, R, \gamma$ and $\gamma'$, $\text{Generate}(\gamma, \varphi_0, \Sigma, R) = (\varphi_1, M)$ and $\text{Generate}(\gamma', \varphi_0, \Sigma, R) = (\varphi_2, M')$, then $\varphi_1 \leq \varphi_2$

**Proof.** From Algorithm [3] let $\varphi_n = \{s\}$.

To show that $\Sigma'; R'; L' \models \{s, s'\}$. From Lemma [3] the conclusion holds.

Theorem 1. Given a $\Sigma, R, L$, $\text{Generate}(\gamma, \varphi_0, \Sigma, R) = (\varphi_1, M)$ and $\text{Generate}(\gamma', \varphi_0, \Sigma, R) = (\varphi_2, M')$, then $\varphi_1 \leq \varphi_2$

**Proof.** Follows by induction on the length of log $L$.
- Ind. case:
  
  IH: $\text{Generate}(\gamma, \varphi_0, \Sigma, R) = (\varphi_1, M)$ and $\text{Generate}(\gamma', \varphi_0, \Sigma, R) = (\varphi_2, M')$, then $\varphi_1 \leq \varphi_2$.
  
  T.S. $\text{Generate}(\gamma', \varphi_0, \Sigma, R) = (\varphi_1, M_1)$ and $\text{Generate}(\gamma', \varphi_0, \Sigma, R) = (\varphi_2, M_1')$, then $\varphi_1 \leq \varphi_2$. From Lemma [5] Lemma [1] and the IH, the conclusion holds.

Theorem 2. Given a log $L$, a typing environment $\Sigma$ and a set of relations $R$, if $\text{Generate}(\{\}, \Sigma, R) = (\varphi, M)$, then $\Sigma; R; L \models \varphi$

**Proof.** The proof is by induction on the length of log $L$.

We omit $M$ for clarity from the definition of Generate. For the base case when the log is empty, $\text{Generate}([\{\}, \Sigma, R] = \{\}$ and $\Sigma; R; \cdot \models \{\}$. For the inductive case from the inductive hypothesis, we have $\text{Generate}([\{\}, \Sigma, R] = \varphi$, then $\Sigma; R; L \models \varphi$.

To show that $\text{Generate}(\{\}, \Sigma, R) = \varphi$, then $\Sigma; R; \gamma, L \models \varphi$. From Algorithm [3] $\varphi_n = \{s\} \cup \varphi_0$. Applying IND, we need to show that $\Sigma; R; \gamma \models \{s\}$.

Let $\gamma = r @ T$ where $r$ includes all actions. From Algorithm [1] and OWN, $\forall a \in (\text{owner}(T, i), T); \Sigma; R; T \vdash a$. Similarly, for Algorithm [4] and RELN, ATTR.

Thus, $\Sigma; R; T \vdash \{a_i(t_i)\}_{i=1}^{n} \cup \{s\} \cup \{s\}$ from Algorithm [3] and PRED, $\Sigma; R; \gamma \models \{s\}$ and the conclusion holds.