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 both 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.

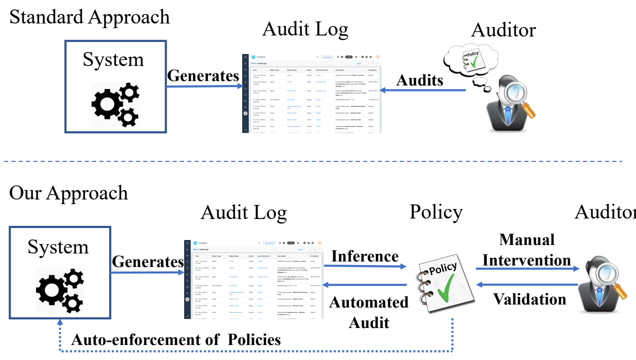

Fig. 1: Standard auditing procedure v/s our approach

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 PersonalInformation.

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 on which the action was performed, when the action was performed 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 may also include entries for data sent out on the network. In the system we consider, a log entry (γ) is a tuple of the form: $\langle A, U, o, \mathit{einfo} \mathbin{@} \tau \rangle$ where A is the action, U is the user, o is the resource, *einfo* is any additional information, and τ 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 σ, 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

$$
\begin{array}{rcll}
\text{Types} & \sigma & := & \texttt{principal} \mid \texttt{data} \\
\text{Terms} & t & := & \texttt{Alice} \mid \texttt{admin} \mid \ldots \\
\text{Actions} & C & := & \texttt{read} \mid \texttt{write} \mid \texttt{send} \\
\text{State Pred.} & P & := & \texttt{has\_attr} \mid \texttt{has\_reln} \mid \texttt{owner} \\
\text{Atoms} & a & := & P\ t_1 \ldots t_n \\
\text{Access Pred.} & r & := & C\ t_1 \ldots t_n \\
\text{Formulas} & s & := & (a_1 \wedge \ldots \wedge a_n) \supset \texttt{may}\ r \mid \forall \vec{x}.s \\
\text{Policy} & \varphi & := & \{s_1, \ldots, s_n\}
\end{array}
$$

Fig. 2: Policy Syntax

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 $\wedge$ is standard while $a \supset \texttt{may}\ r$ indicates that the access r can only happen if a holds. The quantifier $\forall x$ ranges over all possible terms of a particular type σ if x has the type σ. As the domain of every type σ 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_i and the access predicate r. The policy φ is represented in disjunctive normal form over all formulas s_i. 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 $\Sigma; R; \mathcal{L} \vDash \varphi$ states that a log $\mathcal{L}$ satisfies the policy φ under the typing environment Σ 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 φ while rule Ind states that if a log entry γ satisfies the policy φ and a log $\mathcal{L}$ satisfies φ, then the larger log $\gamma, \mathcal{L}$ also satisfies φ. The rule Access states that a log entry γ satisfies a policy φ, if it satisfies any one of the formulas in φ. Rule Inst states that if γ satisfies a formula s instantiated with terms in the log entry γ, it also satisfies the general formula s. The rule Pred says a log entry satisfies an instantiated formula γ, 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 Σ 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

$$\textsc{Attr}\quad \frac{\begin{array}{c} R \vdash [id = t, p_1 = v_1, \ldots, \ldots, p_n = v_n, (T_1, T_2)] \\ T_1 \le T \le T_2 \qquad k \in \{1, \ldots, n\} \end{array}}{\Sigma; R; T \vdash \texttt{has\_attr}\ t\ p_k\ v_k}$$

$$\textsc{Reln}\quad \frac{R \vdash [id = (t_1, t_2), A = r, (T_1, T_2)] \qquad T_1 \le T \le T_2}{\Sigma; R; T \vdash \texttt{has\_reln}\ t_1\ t_2\ r}$$

$$\textsc{Own}\quad \frac{R \vdash [id = t_1, \mathsf{owner} = t_2, (T_1, T_2)] \qquad T_1 \le T \le T_2}{\Sigma; R; T \vdash \texttt{owner}\ t_1\ t_2}$$

$$\textsc{Pred}\quad \frac{\begin{array}{c} \gamma = r(\vec{t})@T \qquad \Sigma; R; T \vdash \{a_i(\vec{t_i})\}_{i=1}^{n} \\ \forall t' \in \vec{t_i}.\big(t' \in \vec{t} \ \vee \ (\exists t_r \in \vec{t}.a_i = \texttt{owner}\ t_r\ t')\big) \end{array}}{\Sigma; R; \gamma \vDash \big\{ \big(\bigwedge_{i=1}^{n} a_i(\vec{t_i}) \big) \supset \texttt{may}\ r(\vec{t}) \big\}}$$

$$\textsc{Inst}\quad \frac{\gamma = r(\vec{t})@T \qquad \vec{t} : \Sigma(\vec{x}) \qquad \Sigma; R; \gamma \vDash \{s[\vec{t}/\vec{x}]\}}{\Sigma; R; \gamma \vDash \{(\forall \vec{x}.s)\}}$$

$$\textsc{Access}\quad \frac{\Sigma; R; \gamma \vDash \{s\}}{\Sigma; R; \gamma \vDash \varphi \cup \{s\}} \qquad \textsc{Ind}\quad \frac{\begin{array}{c} \Sigma; R; \gamma \vDash \varphi \\ \Sigma; R; \mathcal{L} \vDash \varphi \end{array}}{\Sigma; R; \gamma, \mathcal{L} \vDash \varphi} \qquad \textsc{Base}\quad \frac{}{\Sigma; R; \cdot \vDash \varphi}$$

Fig. 3: Policy Enforcement Rules

predicate. We write $R \vdash$ rec to indicate that rec is a valid record in the set of relations R.

Assuming that the sets of principals, resources, roles, relations, predicates and other entities in the system are fixed, the set of formulas (s_i) based on different permutations of these entities is finite. This leads to the system policies having a partial order, $\varphi_1 \le \varphi_2$ meaning that φ_2 is a more permissive policy than φ_1. The policy $\{\}$ represents the most restrictive policy with no formulas while the most permissive policy would contain all possible formulas (or their reduced versions) allowing all accesses.

Definition 1. *Given two policies φ_1 and φ_2, we say that $\varphi_1 \le \varphi_2$ for all logs $\mathcal{L}$, typing environments Σ and sets of relations R, if $\Sigma; R; \mathcal{L} \vDash \varphi_1$, then $\Sigma; R; \mathcal{L} \vDash \varphi_2$*

Lemma 1. *Given two policies φ_1 and φ_2, and a formula s, if $\varphi_1 \le \varphi_2$, then $\varphi_1 \cup \{s\} \le \varphi_2 \cup \{s\}$*

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

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.

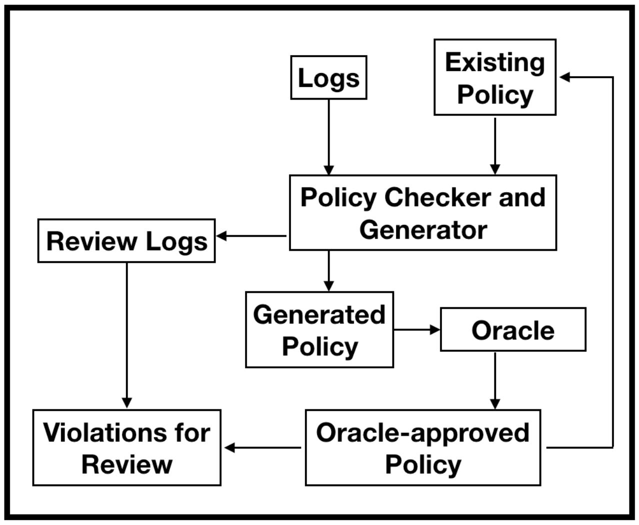

Fig. 4: Policy Inference Framework

```
Input: $R$ - list of relations
       $\vec{t}$ - list of variables
       $T$ - time
Output: $P$ - set of predicates
        $\vec{t'}$ - updated list of variables
Function own ($R$, $\vec{t}$, $T$):
    $P \leftarrow \{\}$; $\vec{t'} \leftarrow \vec{t}$;
    for $i \leftarrow 0$ to $|R|$ do
        $r \leftarrow R[i]$;
        if ownerTuple($r$) $\wedge\ r = (id = t_1, \mathtt{owner} = t_2, (T_1, T_2))$ then
            if $t_1 \in \vec{t}$ and $T_1 \le T \le T_2$ then
                $P \leftarrow P \cup \{\mathtt{owner}\ t_1\ t_2\}$;
                $\vec{t'} \leftarrow \vec{t'} \cup \{t_2\}$;
            end
        end
    end
End Function
```

Algorithm 1: Definition of own$(R, \vec{t}, T)$

```
Input: $R$ - list of relations
       $\vec{t}$ - list of variables
       $T$ - time
Output: $P$ - set of predicates
Function rel ($R$, $\vec{t}$, $T$) :
    $P \leftarrow \{\}$;
    for $i \leftarrow 0$ to $|R|$ do
        $r \leftarrow R[i]$;
        if not ownerTuple($r$) and
         $r = (id = t_1, p = v, (T_1, T_2))$ then
            if $t_1 \in \vec{t}$ and $T_1 \leq T \leq T_2$ then
                $P \leftarrow P \cup \{\texttt{has\_attr}\ t_1\ p\ v\}$;
            end
        else if $r = (id = (t_1, t_2), reln = v, (T_1, T_2))$ then
            if $t_1 \in \vec{t}$ and $t_2 \in \vec{t}$ and $T_1 \leq T \leq T_2$ then
                $P \leftarrow P \cup \{\texttt{has\_reln}\ t_1\ t_2\ v\}$;
            end
        end
    end
End Function
```

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

```
Input: $\mathcal{L}$ - Log
       $R$ - Relations
       $\Sigma$ - typing environment
Output: $\varphi$ - policy
Function infer ($\mathcal{L}$, $R$, $\Sigma$) :
    $\varphi \leftarrow \{\}$;
    for $i \leftarrow 0$ to $|\mathcal{L}|$ do
        $r(\vec{t})@T \leftarrow \mathcal{L}[i]$;
        $(P_o, \vec{t'}) \leftarrow$ own$(R, \vec{t}, T)$;
        $P_r \leftarrow$ rel$(R, \vec{t'}, T)$;
        $P \leftarrow P_o \cup P_r$;
        $\vec{x} \leftarrow \Sigma(\vec{t'})$;
        $s \leftarrow \big(\bigwedge_{p \in P} p\big) \supset \texttt{may}\ r(\vec{t})$;
        $s(\vec{x}) \leftarrow s[\vec{x}/\vec{t'}]$;
        $\varphi \leftarrow \varphi \cup \{\forall \vec{x}. s(\vec{x})\}$;
    end
End Function
```

Algorithm 3: Definition of infer$(\mathcal{L}, R, \Sigma)$

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 ($\mathcal{L}$) along with the set of relations in the database R and the typing environment Σ. 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 terms 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 4) 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 own, 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

$$\begin{aligned}R = [&(id = \text{Alice}, \text{role} = \text{doctor}, (T_1, T_1'));\\ &(id = \text{Charlie}, \text{role} = \text{doctor}, (T_2, T_2'));\\ &(id = \text{Bob}, \text{role} = \text{patient}, (T_3, T_3'));\\ &(id = \text{Bob\_PHI}, \texttt{owner} = \text{Bob}, (T_4, T_4'));\\ &(id = (\text{Alice},\ \text{Bob}), \mathit{reln} = \text{doctor\_of}, (T_5, T_5'));\\ &(id = (\text{Charlie},\ \text{Bob}), \mathit{reln} = \text{doctor\_of}, (T_6, T_6'))]\end{aligned}$$

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:

$$P_1 = \texttt{owner}\ \text{Bob\_PHI Bob} \tag{P1}$$

The set $\vec{t'}$ now includes Bob, i.e.,

$$\vec{t'} = \{\text{Alice}, \text{Bob\_PHI}, \text{Charlie}, \text{Bob}\}$$

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

$$P_2 = \texttt{has\_attr}\ \text{Alice role doctor} \tag{P2}$$

$$P_3 = \texttt{has\_attr}\ \text{Charlie role doctor} \tag{P3}$$

$$P_4 = \texttt{has\_attr}\ \text{Bob role patient} \tag{P4}$$

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

$$P_5 = \texttt{has\_reln}\ \text{Alice Bob doctor\_of} \tag{P5}$$

$$P_6 = \texttt{has\_reln}\ \text{Charlie Bob doctor\_of} \tag{P6}$$

The formula s_1 inferred by infer by using $\vec{x} = \{k, k', k'', p\}$ and substituting k for Alice, k' for Charlie, k'' for Bob and p for Bob_PHI is :

$$\begin{aligned}&\forall k, k', k'' : \texttt{principal}.\forall p : \texttt{phi}.\\ &\quad \texttt{has\_attr}\ k \text{ role doctor } \wedge \texttt{has\_attr}\ k' \text{ role doctor } \wedge\\ &\quad \texttt{owner}\ p\ k'' \ \wedge \texttt{has\_reln}\ k\ k'' \text{ doctor\_of } \wedge\\ &\quad \texttt{has\_reln}\ k'\ k'' \text{ doctor\_of } \wedge \texttt{has\_attr}\ k'' \text{ role patient}\\ &\supset \texttt{may send}\ k\ p\ k' \text{ (purpose:treatment)}\end{aligned} \tag{F1}$$

Instead of checking the complete HIPAA rule with 84 clauses [18] against the log entry, it is easier to audit the specific instance generated by infer.

2) Reduction: As formulas generated 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 $A \vDash C$, then $A \wedge B \vDash C$.

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 Σ, 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

```
Input: $s_1$ - formula
       $s_2$ - formula
       $\Sigma$ - typing environment
Output: $b$ - Boolean result
Function reduce($s_1$, $s_2$, $\Sigma$):
1   $b \leftarrow$ False;
2   $\vec{x} \leftarrow$ bvar($s_1$); $\vec{y} \leftarrow$ bvar($s_2$);
3   $\{a_i^x\}_{i=1}^n \leftarrow$ antt($s_1$); $\{a_i^y\}_{i=1}^m \leftarrow$ antt($s_2$);
4   $r_x(\vec{x_s}) \leftarrow$ consq($s_1$); $r_y(\vec{y_s}) \leftarrow$ consq($s_2$);
5   if $r_x = r_y \wedge |\vec{x_s}| = |\vec{y_s}| \wedge \Sigma(\vec{x_s}) = \Sigma(\vec{y_s})$ then
6       if $|\vec{x}| < |\vec{y}| \wedge n < m$ then
7           $\vec{y'} \leftarrow (\vec{y} \setminus \vec{y_s})$;
8           $\vec{s'} \leftarrow \{a_i^y[\vec{x_s}/\vec{y_s}]\}_{i=1}^m$;
9           $\{\vec{x_i}\}_{i=1}^m \leftarrow$ comb($(\vec{x} \setminus \vec{x_s}), \vec{y'}$);
10          $\{\vec{s_i}\}_{i=1}^m \leftarrow \{\vec{s'}[\vec{x_i}/\vec{y'}]\}_{i=1}^m$;
11          if $\exists i \in \{1..m\}.(\{a_j^x\}_{j=1}^n \subseteq \vec{s_i})$ then
12              $b \leftarrow$ True;
13          end
14      end
15  end
End Function
```

Algorithm 4: Definition of reduce(s_1, s_2, Σ)

(on line 5) if the consequents (access predicates) $r_x(\vec{x}_s)$ and $r_y(\vec{y}_s)$ of the two formulas are the same under the typing environment Σ, and continues if $\vec{x}_s$ is equal in length and type to $\vec{y}_s$ (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 $\vec{y}$. 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:

$$\texttt{send}\ \text{Alice Dave\_PHI Charlie purpose:treatment} \ @\ T' \tag{T2}$$

Suppose

$$\begin{aligned}R = [&(id = \text{Alice}, \text{role} = \text{doctor}, (T_7, T_7'));\\ &(id = \text{Charlie}, \text{role} = \text{doctor}, (T_8, T_8'));\\ &(id = \text{Dave}, \text{role} = \text{patient}, (T_9, T_9'));\\ &(id = \text{Dave\_PHI}, \texttt{owner} = \text{Dave}, (T_{10}, T_{10}'));\\ &(id = (\text{Alice},\ \text{Dave}), \mathit{reln} = \text{doctor\_of}, (T_{11}, T_{11}'))]\end{aligned}$$

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:

$$P_7 = \texttt{owner}\ \text{Dave\_PHI Dave} \tag{P7}$$

The set $\vec{t'}$ now includes Dave, i.e.,

$$\vec{t'} = \{\text{Alice}, \text{Dave\_PHI}, \text{Charlie}, \text{Dave}\}$$

From rel, we have the following predicates:

$$P_8 = \texttt{has\_attr}\ \text{Alice role doctor} \tag{P8}$$

$$P_9 = \texttt{has\_attr}\ \text{Charlie role doctor} \tag{P9}$$

$$P_{10} = \texttt{has\_attr}\ \text{Dave role patient} \tag{P10}$$

$$P_{11} = \texttt{has\_reln}\ \text{Alice Bob doctor\_of} \tag{P11}$$

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

$$\begin{aligned} &\forall k, k', k'' : \texttt{principal}.\forall p : \texttt{phi}. \\ &\quad \texttt{has\_attr}\ k \text{ role doctor } \wedge \texttt{ has\_attr}\ k' \text{ role doctor} \wedge \\ &\quad \texttt{owner}\ p\ k'' \wedge \texttt{ has\_reln}\ k\ k'' \text{ doctor\_of} \wedge \\ &\quad \texttt{has\_attr}\ k'' \text{ role patient} \\ &\supset \texttt{ may send}\ k\ p\ k' \text{ (purpose:treatment)} \end{aligned} \tag{F2}$$

The function reduce takes in the formula above (s_2) and the one generated before, s_1 (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

$$P_{12} = \texttt{has\_attr}\ \text{Alice role director} \tag{P12}$$

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 ($\mathcal{L}$), an existing policy (φ_0), a typing environment (Σ) and a set of relations (R), Generate starts by inferring a formula for every log-entry in $\mathcal{L}$ 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.

```
Input: $\mathcal{L}$ - a list of log entries
       $\varphi_o$ - existing policy
       $\Sigma$ - typing environment
       $R$ - relational database
Output: $\varphi_n$ - inferred policy
        $M$ - map from formula to set of formulas
Function Generate ($\mathcal{L}$, $\varphi_o$, $\Sigma$, $R$):
    $\varphi_o \leftarrow$ infer($\mathcal{L}, R, \Sigma$); $\varphi_n \leftarrow \varphi_o$; $M \leftarrow []$;
    foreach $s_i \in \varphi_o$ do
        foreach $s_j \in \varphi_o \setminus \{s_i\}$ do
            if reduce($s_i, s_j, \Sigma$) then
                $\varphi_n \leftarrow \varphi_n \setminus s_j$;
                $M \leftarrow$ appendMap($M, s_i, s_j$);
            else if reduce($s_j, s_i, \Sigma$) then
                $\varphi_n \leftarrow \varphi_n \setminus s_i$;
                $M \leftarrow$ appendMap($M, s_j, s_i$);
                break;
            end
        end
    end
End Function
```

Algorithm 5: Policy Inference Algorithm

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 φ 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 Σ, a set of relations R, a log $\mathcal{L}$, and formulas s and s', if $\mathcal{L}$ satisfies either the formula s or s', then it satisfies the policy containing both s and s'.*

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

Lemma 5. *Given a typing environment Σ, a set of relations R, and two log entries γ and γ', if our algorithm infers the* *policy φ_1 for γ and φ_2 for γ, γ', then $\varphi_1 \leq \varphi_2$.*

Theorem 1. *Given a typing environment Σ and a set of relations R, if our algorithm infers a policy φ_1 for the log $\mathcal{L}$ and infers a policy φ_2 for the log $\gamma, \mathcal{L}$, then $\varphi_1 \leq \varphi_2$*

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 φ_o, 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 Σ and a set of relations R, if our algorithm generates a policy φ, then the log $\mathcal{L}$ satisfies φ.*

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 $\mathcal{L}$, typing environment Σ and a set of relations R, and any policy φ', if $\forall \gamma \in \mathcal{L}.\mathsf{Generate}([\gamma], \{\}, \Sigma, R) = (\{s\}, [])$ and $\Sigma; R; \gamma \vDash \varphi'$, then $\exists s' \in \varphi'.(s = s' \vee \mathsf{reduce}(s', s, \Sigma) = \mathsf{True}$*

Proof. Let $\gamma = r(\vec{t})@T$. From Theorem 2, we have $\Sigma; R; \gamma \vDash \{s\}$ such that s contains **all** ownerships, attributes and relationships related to $\vec{t}$. From Lemma 2, $\exists s \in \varphi'.\Sigma; R; \gamma \vDash \{s'\}$. From PRED, we know that s' contains ownerships, attributes and relationships related to $\vec{t}$. As R remains the same, s' may contain all or a subset of state predicates present in s. Thus, either $s = s'$ or reduce, we have that $\mathsf{reduce}(s', s, \Sigma) = \mathsf{True}$ □

```
Input: $\varphi$ - policy
       $M$ - map of formulas to list of formulas
Output: $\varphi_a$ - approved policy
Function oracle($\varphi$, $M$):
    $\varphi_a \leftarrow \{\}$; $\varphi_m \leftarrow \{\}$;
    foreach $s \in \varphi$ do
        if isValid($s$) then
            $\varphi_a \leftarrow \{s\} \cup \varphi_a$;
        else
            $\varphi_s \leftarrow M(s) \cup \varphi_s$;
        end
    end
    foreach $s \in \varphi_s$ do
        if isValid($s$) then
            $\varphi_a \leftarrow \{s\} \cup \varphi_a$;
        end
    end
End Function
```

Algorithm 6: Definition of $\mathsf{oracle}(\varphi, M)$

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

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: $\forall \Sigma'; R'; \mathcal{L}'.\Sigma'; R'; \mathcal{L}' \vDash \varphi \implies \Sigma'; R'; \mathcal{L}' \vDash \varphi'$.
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 φ'. 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 function takes in the generated policy φ along with the map M and returns the approved policy φ_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 4). 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 3). 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 the

$$\textsc{Inst}\ \dfrac{F1 = s_1(\vec{x}) \qquad \vec{t} = \{\text{Alice, Charlie, Bob, Bob\_PHI}\} \qquad \textsc{Pred}\ \dfrac{\textsc{Attr, Own, Reln}\ \dfrac{\cdots}{\Sigma; R; T \vdash \{P_i\}_{i=1}^{6}}}{\Sigma; R; \gamma \vDash \{s_1(\vec{t})\}}}{\Sigma; R; (\gamma = \texttt{send}\ \text{Alice Bob\_PHI Charlie purpose:treatment}\ @T) \vDash \{F1\}}$$

Fig. 5: Validation of transaction (T1)

$$\textsc{Inst}\ \dfrac{F1 = s_1(\vec{x}) \qquad \vec{t} = \{\text{Alice, Charlie, Dave, Dave\_PHI}\} \qquad \textsc{Pred}\ \dfrac{\dfrac{\lightning}{\Sigma; R; T' \vdash \texttt{has\_reln}\ \text{Charlie Dave doctor\_of}} \qquad \textsc{Attr, Own, Reln}\ \dfrac{\cdots}{\Sigma; R; T' \vdash \{P_i\}_{i=7}^{11}}}{\Sigma; R; \gamma \vDash \{s_1(\vec{t})\}}}{\Sigma; R; (\gamma = \texttt{send}\ \text{Alice Dave\_PHI Charlie purpose:treatment}\ @T') \vDash \{F1\}}$$

Fig. 6: Validation of transaction (T2)

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 (F1) is a valid formula in the policy but s_2 (F2) 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, $\mathsf{isValid}(s_2)$ 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 3. For every log entry $\gamma = \langle A, U, o, \mathit{einfo}@T\rangle$, the enforcement checks if γ satisfies the approved policy φ_a. If $\Sigma; R; \langle A, U, o, \mathit{einfo}@T\rangle \vDash \varphi_a$, 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 T1 as shown in Figure 5. The enforcement starts by using the INST rule by replacing $\vec{x}$ in s_1 with terms $\vec{t}$. 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 (T1). The derivation uses P_i that correspond to the predicates P1 to P6 defined above in the example for s_1. Figure 6 attempts to validate transaction (T2). It proceeds in a similar fashion as transaction (T1) using the predicates P7 to P11 but does not find a relevant entry for establishing `has_reln` Charlie Dave doctor_of in R thereby reporting the transaction as a violation ($\lightning$).

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 Σ 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 Σ 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 [1] places restriction on how this information can be accessed and/or disclosed to a third-party by a user (in this case, a covered entity). 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* [33], 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 $s_1 = (a_1 \wedge a_2) \supset \texttt{may}\ r$, $s_2 = (a_1 \wedge a_2 \wedge a_3) \supset \texttt{may}\ r$ and $s_3 = (a_1 \wedge a_2 \wedge a_4) \supset \texttt{may}\ r$, 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

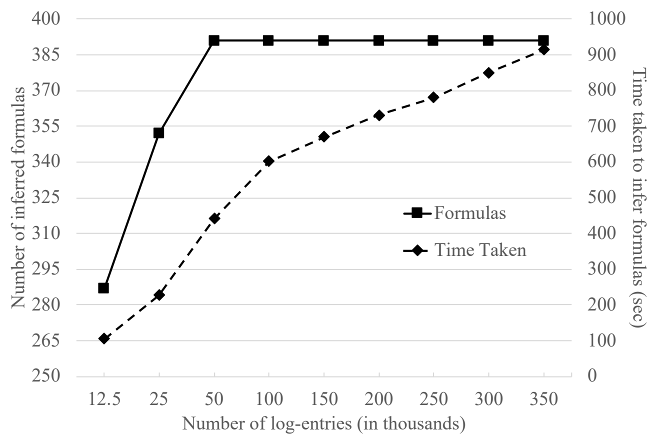

Fig. 7: Number of inferred formulas and the time taken to infer them for different number of log entries

graph for the number of formulas inferred and time taken for inference for different number of log entries. Note that the logs did not contain any violations.

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 $\S164.508(a)(2)$ 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:

$$\begin{aligned}
&\forall k, k', k'' : \texttt{principal}.\forall p : \texttt{phi}.\\
&\quad \texttt{has\_attr}\ k\ \text{role c-entity}\ \wedge\ \texttt{has\_attr}\ k'\ \text{role c-entity} \wedge\\
&\quad \texttt{owner}\ p\ k''\ \wedge\ \texttt{has\_reln}\ k\ k''\ \text{hospital\_of} \wedge\\
&\quad \texttt{has\_reln}\ k'\ k''\ \text{health\_care\_provider\_of} \wedge\\
&\quad \texttt{has\_attr}\ k''\ \text{role patient}\\
&\supset\ \texttt{may send}\ k\ p\ k'\ \text{(purpose:payment)}
\end{aligned}$$

The actual HIPAA policy ($\S164.506(c)(3)$) 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,

$$\begin{aligned}
&\forall k, k', k'' : \texttt{principal}.\forall p : \texttt{phi}.\\
&\quad \texttt{has\_attr}\ k\ \text{role c-entity}\ \wedge\ \texttt{has\_attr}\ k'\ \text{role c-entity} \wedge\\
&\quad \texttt{owner}\ p\ k''\ \wedge\ \texttt{has\_reln}\ k\ k''\ \text{hospital\_of} \wedge\\
&\quad \texttt{has\_attr}\ k''\ \text{role patient}\\
&\supset\ \texttt{may send}\ k\ p\ k'\ \text{(purpose:payment)}
\end{aligned}$$

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,

$$\begin{aligned}
&\forall k, k', k'' : \texttt{principal}.\forall p : \texttt{phi}.\\
&\quad \texttt{has\_attr}\ k\ \text{role c-entity}\ \wedge\ \texttt{has\_attr}\ k'\ \text{role c-entity} \wedge\\
&\quad \texttt{owner}\ p\ k''\ \wedge\ \texttt{has\_reln}\ k\ k''\ \text{hospital\_of} \wedge\\
&\quad \texttt{has\_attr}\ k''\ \text{role patient}\\
&\supset\ \texttt{may send}\ k\ p\ k'\ \text{(purpose:payment)}
\end{aligned} \tag{1}$$

$$\ldots \supset\ \texttt{may send}\ k\ p\ k'\ \text{(purpose:treatment)} \tag{2}$$

$$\ldots \supset\ \texttt{may send}\ k\ p\ k'\ \text{(purpose:healthcare-operations)} \tag{3}$$

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

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

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.

VII. Discussion

A. Negations and existential formulas

While our policy logic is a fragment of first-order logic without existential quantifiers and negations, it is sufficiently powerful to express access formulas. As access is allowed only if one of the inferred formulas holds; all other formulas are implicitly assumed to not hold at the time of access indicating possible violations. The domains of various types are bounded; thus, negations of atomic predicates are well-defined sets of *satisfying* atomic predicates. Our algorithm does not infer negations in formulas as those predicates should "not" be true for the formula to hold as the logs do not contain information about what should not be true at the time of access. Instead, it starts by assuming that all entries in the audit logs are correct; hence, inferring positive formulas that must hold for the entry to be generated.

For instance, consider an organization that has three user roles - doctor, patient and nurse. Then, the predicate $\neg\texttt{has\_attr}(k, \text{role}, \text{doctor})$ is equivalent to $\texttt{has\_attr}(k, \text{role}, \text{patient}) \vee \texttt{has\_attr}(k, \text{role}, \text{nurse})$. If a formula requires $\neg\texttt{has\_attr}(k, \text{role}, \text{doctor})$ and an access was *granted* to k, then either $\texttt{has\_attr}(k, \text{role}, \text{patient})$ was true or $\texttt{has\_attr}(k, \text{role}, \text{nurse})$ was true. Based on the log entry (the role of k), either one of these formulas is inferred by our algorithm. Once both are added to the policy through different formulas (in disjunctive normal form), they represent the equivalent predicate $\neg\texttt{has\_attr}(k, \text{role}, \text{doctor})$. The inferred policy might not be the actual policy but an equivalent policy given the current domains of the types.

For existential operators, the algorithm generates formulas with specific values as it is difficult to anticipate whether it is an or what the existential formula is that a log-entry satisfies; this is more restrictive than the original clause which requires "some" value. The algorithm generates more possible values used in practice as more log entries are parsed. For instance, DeYoung et al. [18] formalize the clause §$164.506(c)(4)$ of HIPAA as:

$$\begin{aligned} &\mathit{role}(p1, \mathit{centity}) \wedge \mathit{role}(p2, \mathit{centity}) \wedge (t \in \mathit{phi}) \wedge \\ &(\exists r1 : \mathit{rel.}\ \mathit{inrelationship}(p1, r1, q) \wedge \mathit{pertainsto}(t, r1)) \wedge \\ &(\exists r2 : \mathit{rel.}\ \mathit{inrelationship}(p2, r2, q) \wedge \mathit{pertainsto}(t, r2)) \wedge \\ &((u \in \mathit{healthcare\_fraud\_abuse\_detection}) \vee \\ &(u \in \mathit{healthcare\_fraud\_abuse\_compliance})) \end{aligned}$$

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.

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 [24] 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 Masoumzadeh [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.* [45] present one of the earliest approaches to mining program specifications from execution traces using machine learning algorithms. IPM2 [46] performed interaction-pattern mining to discover the usage behavior of users. A similar work uses usage scenarios to mine API specifications from source code [47]. Mining temporal invariants and properties [48–50] have also been a subject of active research over the past few years. Other mining approaches have targeted parametric specifications [51], scenario-based specifications [52–54] object usage [55] and software behaviors [56].

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

[1] Office for Civil Rights (OCR), “Summary of the HIPAA Privacy Rule,” https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html, 2013.

[2] HHS Press Office, “$5.5 million HIPAA settlement shines light on the importance of audit controls,” https://www.hhs.gov/about/news/2017/02/16/hipaa-settlement-shines-light-on-the-importance-of-audit-controls.html, 2017.

[3] “Human error still the cause of many data breaches,” https://www.helpnetsecurity.com/2019/06/17/human-error-data-breach/, 2019.

[4] “HIPAA Access Logs Audits,” https://www.hipaaformsps.com/hipaa-access-logs-audits/, 2019.

[5] “A Compliance Primer for IT Professionals,” https://www.sans.org/reading-room/whitepapers/compliance/compliance-primer-professionals-33538, 2020.

[6] US Congress, “Gramm-Leach-Bliley Act, Financial Privacy Rule,” http://www.ftc.gov/privacy/glbact/glbsub1.htm, 1999.

[7] Mark Keppler, “How Do Internal Audits Work?” https://www.ispartnersllc.com/blog/how-do-internal-audits-work/, 2019.

[8] Travis Good, “What is the cost of a HIPAA audit?” https://datica.com/blog/what-is-the-cost-of-a-hipaa-audit/, 2015.

[9] Nitasha Tiku, “Facebook’s 2017 Privacy Audit Didn’t Catch Cambridge Analytica,” https://www.wired.com/story/facebooks-2017-privacy-audit-didnt-catch-cambridge-analytica/, 2018.

[10] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, “A calculus for access control in distributed systems,” *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 4, p. 706–734, Sep. 1993. [Online]. Available: https://doi.org/10.1145/155183.155225

[11] Deepak Garg and F. Pfenning, “Non-interference in constructive authorization logic,” in *19th IEEE Computer Security Foundations Workshop (CSFW’06)*, July 2006, pp. 11 pp.–296.

[12] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini, “Audit-based compliance control,” *International Journal of Information Security*, vol. 6, no. 2, pp. 133–151, 2007. [Online]. Available: https://doi.org/10.1007/s10207-007-0017-y

[13] S. Etalle and W. H. Winsborough, “A posteriori compliance control,” in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT ’07, 2007, p. 11–20. [Online]. Available: https://doi.org/10.1145/1266840.1266843

[14] D. Garg, L. Jia, and A. Datta, “Policy Auditing over Incomplete Logs: Theory, Implementation and Applications,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS ’11, 2011, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046726

[15] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu, “Monitoring compliance policies over incomplete and disagreeing logs,” in *Runtime Verification*, S. Qadeer and S. Tasiran, Eds., 2012, pp. 151–167.

[16] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic, “Evidence-based audit,” in *2008 21st IEEE Computer Security Foundations Symposium*, June 2008, pp. 177–191.

[17] J. Reuben, L. A. Martucci, and S. Fischer-Hübner, *Automated Log Audits for Privacy Compliance Validation: A Literature Survey*, 2016, pp. 312–326. [Online]. Available: https://doi.org/10.1007/978-3-319-41763-9_21

[18] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta, “Experiences in the logical specification of the hipaa and glba privacy laws,” in *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*, ser. WPES ’10, 2010, p. 73–82. [Online]. Available: https://doi.org/10.1145/1866919.1866930

[19] D. Basin, F. Klaedtke, and S. Müller, “Policy monitoring in first-order temporal logic,” in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., 2010, pp. 1–18.

[20] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” *SIGMOD Rec.*, vol. 29, no. 2, p. 1–12, May 2000. [Online]. Available: https://doi.org/10.1145/335191.335372

[21] L. Bauer, S. Garriss, and M. K. Reiter, “Detecting and resolving policy misconfigurations in access-control systems,” *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1952982.1952984

[22] Z. Xu and S. D. Stoller, “Algorithms for mining meaningful roles,” in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012, pp. 57–66.

[23] ——, “Mining parameterized role-based policies,” in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’13, 2013, pp. 255–266. [Online]. Available: https://doi.org/10.1145/2435349.2435384

[24] ——, “Mining attribute-based access control policies from logs,” in *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy XXVIII - Volume 8566*, ser. DBSec 2014, 2014, p. 276–291. [Online]. Available: https://doi.org/10.1007/978-3-662-43936-4_18

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

[26] D. Mocanu, F. Turkmen, and A. Liotta, “Towards abac policy mining from logs with deep learning,” in *In proc. of the 18th International Multiconference, IS 2015, Intelligent Systems*, 10 2015.

[27] T. Bui, S. D. Stoller, and J. Li, “Mining relationship-based access control policies,” in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT ’17 Abstracts, 2017, p. 239–246. [Online]. Available: https://doi.org/10.1145/3078861.3078878

[28] P. Iyer and A. Masoumzadeh, “Mining positive and negative attribute-based access control policy rules,” in *Proceedings of the 23nd ACM on Symposium on Access Control Models and*

Technologies, ser. SACMAT '18, 2018, p. 161–172. [Online]. Available: https://doi.org/10.1145/3205977.3205988

[29] C. Cotrini, T. Weghorn, and D. Basin, "Mining abac rules from sparse logs," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, April 2018, pp. 31–46.

[30] T. Bui, S. D. Stoller, and H. Le, "Efficient and extensible policy mining for relationship-based access control," in *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '19, 2019, p. 161–172. [Online]. Available: https://doi.org/10.1145/3322431.3325106

[31] M. W. Sanders and C. Yue, "Mining least privilege attribute based access control policies," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, 2019, p. 404–416. [Online]. Available: https://doi.org/10.1145/3359789.3359805

[32] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications." *Pacific J. Math.*, vol. 5, no. 2, pp. 285–309, 1955. [Online]. Available: https://projecteuclid.org:443/euclid.pjm/1103044538

[33] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, 2002, p. 143–154.

[34] R. Corin, S. Etalle, J. den Hartog, G. Lenzini, and I. Staicu, "A logic for auditing accountability in decentralized systems," in *Formal Aspects in Security and Trust*, T. Dimitrakos and F. Martinelli, Eds., 2005, pp. 187–201.

[35] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a theory of accountability and audit," in *Proceedings of the 14th European Conference on Research in Computer Security*, ser. ESORICS'09, 2009, p. 152–167.

[36] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Correct audit logging: Theory and practice," in *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635*, 2016, p. 139–162.

[37] D. Fabbri and K. LeFevre, "Explanation-based auditing," *Proc. VLDB Endow.*, vol. 5, no. 1, p. 1–12, Sep. 2011. [Online]. Available: https://doi.org/10.14778/2047485.2047486

[38] O. Chowdhury, L. Jia, D. Garg, and A. Datta, "Temporal mode-checking for runtime monitoring of privacy policies," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., 2014, pp. 131–149.

[39] A. Ene, W. Horne, N. Milosavljevic, P. Rao, R. Schreiber, and R. E. Tarjan, "Fast exact and heuristic methods for role minimization problems," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '08, 2008, p. 1–10. [Online]. Available: https://doi.org/10.1145/1377836.1377838

[40] 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*, ser. SACMAT '08, 2008, p. 21–30. [Online]. Available: https://doi.org/10.1145/1377836.1377840

[41] 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*, ser. CCS '09, 2009, p. 101–111. [Online]. Available: https://doi.org/10.1145/1653662.1653675

[42] 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*, ser. SACMAT '12, 2012, p. 45–56. [Online]. Available: https://doi.org/10.1145/2295136.2295145

[43] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94, 1994, p. 487–499.

[44] M. W. Sanders and C. Yue, "Minimizing privilege assignment errors in cloud services," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18, 2018, p. 2–12. [Online]. Available: https://doi.org/10.1145/3176258.3176307

[45] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02, 2002, p. 4–16. [Online]. Available: https://doi.org/10.1145/503272.503275

[46] M. El-Ramly, E. Stroulia, and P. Sorenson, "From run-time behavior to usage scenarios: An interaction-pattern mining approach," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02, 2002, p. 315–324. [Online]. Available: https://doi.org/10.1145/775047.775095

[47] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, 2007, p. 25–34. [Online]. Available: https://doi.org/10.1145/1287624.1287630

[48] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal api rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, p. 282–291. [Online]. Available: https://doi.org/10.1145/1134285.1134325

[49] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, p. 267–277. [Online]. Available: https://doi.org/10.1145/2025113.2025151

[50] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015, p. 81–92. [Online]. Available: https://doi.org/10.1109/ASE.2015.71

[51] C. Lee, F. Chen, and G. Roundefinedu, "Mining parametric specifications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, p. 591–600. [Online]. Available: https://doi.org/10.1145/1985793.1985874

[52] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007, p. 465–468. [Online]. Available: https://doi.org/10.1145/1321631.1321710

[53] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, p. 387–396. [Online]. Available: https://doi.org/10.1145/1858996.1859081

[54] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 443–453.

[55] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, 2009, p. 371–382. [Online]. Available: https://doi.org/10.1109/ASE.2009.60

[56] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, p. 501–510. [Online]. Available: https://doi.org/10.1145/1368088.1368157

Appendix

A. Lemmas and Proofs

Lemma 1. *For all policies φ_1 and φ_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) $\forall \Sigma, R, \mathcal{L}.\ \Sigma; R; \mathcal{L} \vDash \varphi_1 \implies \Sigma; R; \mathcal{L} \vDash \varphi_2$.
Suppose that $\forall \Sigma', R', \mathcal{L}'.\Sigma'; R'; \mathcal{L}' \vDash \varphi_1 \cup \{s\}$.
T.S.: $\Sigma'; R'; \mathcal{L}' \vDash \varphi_2 \cup \{s\}$.
Proof follows by induction on the length of the log $\mathcal{L}'$ and then applying ACCESS along with (1). □

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

Proof. Proof follows by induction on the length of the log $\mathcal{L}$.
Base case : $\mathcal{L} = [\gamma]$
From IND, $\Sigma; R; \gamma \vDash \varphi$. From ACCESS, we have $\Sigma; R; \gamma \vDash \{s\}$ such that $s \in \varphi$. Hence, the conclusion holds.
Ind. case: $\mathcal{L}' = \gamma', \mathcal{L}$

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

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

Proof. By induction on length of log $\mathcal{L}$.

- Base case: $|\mathcal{L}| = 1$. From ACCESS, and $\Sigma; R; \mathcal{L} \vDash \{s\}$, we have $\Sigma; R; \mathcal{L} \vDash \{s, s'\}$.
- Ind. case: IH: If $\Sigma; R; \mathcal{L} \vDash \{s\}$, then $\Sigma; R; \mathcal{L} \vDash \{s, s'\}$
  T.S. If $\Sigma; R; \gamma, \mathcal{L} \vDash \{s\}$, then $\Sigma; R; \gamma, \mathcal{L} \vDash \{s, s'\}$
  From IND, $\Sigma; R; \gamma \vDash \{s\}$ and $\Sigma; R; \mathcal{L} \vDash \{s\}$.
  From ACCESS, we know that if $\Sigma; R; \gamma \vDash \{s\}$, then $\Sigma; R; \gamma \vDash \{s, s'\}$
  From IH, if $\Sigma; R; \mathcal{L} \vDash \{s\}$, then $\Sigma; R; \mathcal{L} \vDash \{s, s'\}$
  Thus, from IND, the conclusion holds.

$\square$

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

Proof. By induction on length of log $\mathcal{L}$.

- Base case: $|\mathcal{L}| = 1$. Let $r(\vec{t})@T = \mathcal{L}$, $s = \big(\bigwedge_{i=1}^{n} a_i(\vec{x_i})\big) \supset \texttt{may}\ r$ and $s' = \big(\bigwedge_{j=1}^{m} a'_j(\vec{y_j}')\big) \supset \texttt{may}\ r$.
  From Algorithm 4, we know that $r(\vec{x}) = r'(\vec{y}')$ and $\forall i \in \{1, \ldots, m\}. a'_j(\vec{y_j}') \in \{a_i(\vec{x_i})\}_{i=1}^{n}$ such that $m \leq n$. From PRED and $\Sigma; R; r@T \vDash \{s\}$, we know $\Sigma; R; T \vdash \{a_j(\vec{t_j})\}_{j=1}^{n}$. As $\forall i \in \{1, \ldots, m\}. a'_i(\vec{y_i}') \in \{a_j(\vec{x_j})\}_{j=1}^{n}$, from PRED, we have $\Sigma; R; r@T \vDash \{s'\}$. Hence, the conclusion holds.
- Ind. case: IH: If $\Sigma; R; \mathcal{L} \vDash \{s\}$ and $\mathsf{reduce}(s', s, \Sigma) = \mathsf{True}$, then $\Sigma; R; \mathcal{L} \vDash \{s'\}$
  T.S. If $\Sigma; R; \gamma, \mathcal{L} \vDash \{s\}$ and $\mathsf{reduce}(s', s, \Sigma) = \mathsf{True}$, then $\Sigma; R; \gamma, \mathcal{L} \vDash \{s, s'\}$
  From IND, $\Sigma; R; \gamma \vDash \{s\}$ and $\Sigma; R; \mathcal{L} \vDash \{s\}$.
  From similar reasoning as in base case, we know that if $\Sigma; R; \gamma \vDash \{s\}$, then $\Sigma; R; \gamma \vDash \{s'\}$
  From IH, if $\Sigma; R; \mathcal{L} \vDash \{s\}$, then $\Sigma; R; \mathcal{L} \vDash \{s'\}$
  Thus, from IND, the conclusion holds.

$\square$

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

Proof. From Algorithm 3, let $\varphi_n = \{s\}$. Then for γ, γ', $\varphi_n = \{s, s'\}$. Two cases arise:

- Suppose $\varphi_1 = \{s\}$ and $\varphi_2 = \{s, s'\}$.
  Assume $\forall \Sigma', R', \mathcal{L}'$, $\Sigma'; R'; \mathcal{L}' \vDash \{s\}$
  To show that $\Sigma'; R'; \mathcal{L}' \vDash \{s, s'\}$. From Lemma 3, the conclusion holds.
- Let $\varphi_1 = \{s\}$ and $\varphi_2 = \{s'\}$ and $M' = [(s' \mapsto \{s\})]$.
  Assume $\forall \Sigma', R', \mathcal{L}'$, $\Sigma'; R'; \mathcal{L}' \vDash \{s\}$. From Lemma 4, we have $\Sigma'; R'; \mathcal{L}' \vDash \{s'\}$. Hence, the conclusion holds.

$\square$

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

Proof. Follows by induction on the length of log $\mathcal{L}$.

- Base case: $|\mathcal{L}| = 1$. By Lemma 5, the conclusion holds.
- Ind. case:
  IH: $\mathsf{Generate}(\mathcal{L}, \varphi_0, \Sigma, R) = (\varphi_1, M)$ and $\mathsf{Generate}((\gamma, \mathcal{L}), \varphi_0, \Sigma, R) = (\varphi_2, M')$, then $\varphi_1 \leq \varphi_2$.
  T.S.: $\mathsf{Generate}(\gamma', \mathcal{L}, \varphi_0, \Sigma, R) = (\varphi'_1, M_1)$ and $\mathsf{Generate}((\gamma', \gamma, \mathcal{L}), \varphi_0, \Sigma, R) = (\varphi'_2, M'_1)$, then $\varphi'_1 \leq \varphi'_2$. From Lemma 5, Lemma 1 and the IH, the conclusion holds.

$\square$

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

Proof. The proof is by induction on the length of log $\mathcal{L}$.
We omit M for clarity from the definition of Generate. For the base case when the log is empty, $\mathsf{Generate}([], \{\}, \Sigma, R) = \{\}$ and $\Sigma; R; \cdot \vDash \{\}$.
For the inductive case from the inductive hypothesis, we have $\mathsf{Generate}(\mathcal{L}, \{\}, \Sigma, R) = \varphi$, then $\Sigma; R; \mathcal{L} \vDash \varphi$.
To show that if $\mathsf{Generate}((\gamma, \mathcal{L}), \{\}, \Sigma, R) = \varphi$, then $\Sigma; R; (\gamma, \mathcal{L}) \vDash \varphi$. From Algorithm 3, $\varphi_n = \{s\} \cup \varphi_o$.
Applying IND, we need to show that $\Sigma; R; \gamma \vDash \{s\}$.
Let $\gamma = r@T$ where r includes all actions. From Algorithm 1 and OWN, $\forall a \in (\texttt{owner}(R, \vec{t}, T)). \Sigma; R; T \vdash a$. Similarly, for Algorithm 2 and RELN, ATTR.
Thus, $\Sigma; R; T \vdash \{a_i(\vec{t_i})\}_{i=1}^{n}$ and $s = \bigwedge_{i=1}^{n} a_i(\vec{t_i}) \supset \texttt{may}\ r(\vec{t})$.
From Algorithm 3 and PRED, $\Sigma; R; \gamma \vDash \{s\}$ and the conclusion holds. $\square$