Policy Auditing over Incomplete Logs: Theory, Implementation and Applications

Deepak Garg
Carnegie Mellon University
Pittsburgh, USA
dg@cs.cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, USA
liminjia@cmu.edu

Anupam Datta
Carnegie Mellon University
Pittsburgh, USA
danupam@cmu.edu

ABSTRACT

We present the design, implementation and evaluation of an algorithm that checks audit logs for compliance with privacy and security policies. The algorithm, which we name reduce, addresses two fundamental challenges in compliance checking that arise in practice. First, in order to be applicable to realistic policies, reduce operates on policies expressed in a first-order logic that allows *restricted quantification* over infinite domains. We build on ideas from logic programming to identify the restricted form of quantified formulas. The logic can, in particular, express all 84 disclosure-related clauses of the HIPAA Privacy Rule, which involve quantification over the infinite set of messages containing personal information. Second, since audit logs are inherently *incomplete* (they may not contain sufficient information to determine whether a policy is violated or not), reduce proceeds iteratively: in each iteration, it provably checks as much of the policy as possible over the current log and outputs a residual policy that can only be checked when the log is extended with additional information. We prove correctness, termination, time and space complexity results for reduce. We implement reduce and optimize the base implementation using two heuristics for database indexing that are guided by the syntactic structure of policies. The implementation is used to check simulated audit logs for compliance with the HIPAA Privacy Rule. Our experimental results demonstrate that the algorithm is fast enough to be used in practice.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal logic*; H.2.1 [**Database Management**]: Logical Design; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods; K.4.3 [**Computers and Society**]: Organizational Impacts—*Automation*

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

General Terms

Experimentation, Legal Aspects, Security, Theory

Keywords

Audit, Formal logic, Incomplete logs, Privacy policy

1. INTRODUCTION

Organizations, such as hospitals, banks, and universities, that collect, use, and share personal information have to ensure that they do so in a manner that respects the privacy of the information's subjects. In fact, designing effective processes to ensure compliance with internal policies and privacy regulations, such as the Health Insurance Portability and Accountability Act (HIPAA) [34] and the Gramm-Leach-Bliley Act (GLBA) [33], has become one of the greatest challenges facing organizations today (see, for example, a survey from Deloitte and the Ponemon Institute [13]). This paper presents an approach to address this challenge.

We design and implement an algorithm that checks audit logs for compliance with a rich class of privacy and security policies. The algorithm, which we name reduce, addresses two fundamental challenges in compliance checking that arise in practice. First, in order to be applicable to realistic policies, reduce operates on policies expressed in a first-order logic that allows *restricted quantification* over infinite domains. We design this logic by starting from our prior work on a logic for specifying privacy regulations [15], and restricting the syntax of quantifiers in order to ensure that reduce terminates. In more detail, we note that many HIPAA clauses are of the form $\forall p_1, p_2, m.(\texttt{send}(p_1, p_2, m) \supset \varphi)$ where p_1 and p_2 are principals and m is a message. This formula quantifies over the infinite set of messages, so if an enforcement algorithm were to blindly instantiate the quantifiers with all possible values in the domain, it would not terminate. However, since the number of messages transmitted from a hospital is finite, the predicate $\texttt{send}(p_1, p_2, m)$ is true for only a finite number of substitutions for the variable m (and similarly for p_1 and p_2). At a technical level, we use the idea of *mode checking* from logic programming [2] to ensure that the number of relevant substitutions for every quantified variable is always finite, thus ensuring that reduce terminates. The resulting logic is more expressive than prior logics used for compliance-checking, including propositional temporal logics [6, 18] and first-order metric temporal logic [9], and, in contrast to these logics, can express all 84 disclosure-related clauses in the HIPAA Privacy Rule.

Second, a significant challenge in automated compliance checking is that logs maintained by organizations may be *incomplete*, i.e., they may not contain enough information to decide whether or not the policy has been violated. There are many different sources of incompleteness. Specifically, privacy regulations often permit disclosures based on subjective beliefs (e.g., allowing a hospital to share health information with law enforcement officers if the hospital believes that a death could have been the result of a criminal act) and future obligations (e.g., requiring organizations to notify customers within a prescribed time period if a data breach has occurred). These two classes of policies (which abound in privacy regulations) illustrate why we cannot hope to have an automated enforcement mechanism that decides whether a disclosure is permitted or not at the time the disclosure occurs—in other words, why a preventive enforcement regime is not sufficient for enforcement of such policies. In addition to these inherent forms of incompleteness, sometimes not all relevant information is recorded in a single log, i.e., logs may be spatially distributed. As an important contribution, we observe that such incomplete logs can be abstractly represented as three-valued, *partial structures* that map each atomic formula to either true, false, or unknown [11, 19]. We define the semantics of our logic over such structures. We design reduce to work with partial structures, thus providing a uniform method of compliance checking that accounts for different sources of incompleteness in audit logs. Since logs evolve over time by gathering more information, reduce proceeds iteratively: In each iteration, it checks as much of the policy as possible over the current log and outputs a residual policy that can only be checked when the log is extended with additional information.

We formally prove the following properties of reduce, using $\mathsf{reduce}(\mathcal{L}, \varphi) = \varphi'$ to denote one iteration of reduce: (1) The residual policy φ' output by reduce is *minimal*: it only contains predicates whose truth value is unknown in the current partial structure $\mathcal{L}$; (2) reduce *terminates*: as noted earlier, the finite substitution property of variables quantified over infinite domains is crucial for termination; (3) reduce is *correct*: any extension of $\mathcal{L}$ satisfies the policy φ if and only if it satisfies the residual formula φ'; (4) Assuming that finding an entry in the audit log takes unit time, reduce runs in time polynomial in the size of the audit log where the degree of the polynomial is the size of the policy formula (i.e., $\text{TIME}(|\mathcal{L}|^{O(|\varphi|)})$), and uses space that is polynomial in the size of the policy formula (i.e., $\text{PSPACE}(|\varphi|)$).

We implement reduce and optimize the base implementation with heuristics for database indexing that are guided by the syntactic structure of policies. The implementation is used to check simulated audit logs for compliance with the HIPAA Privacy Rule. Our experimental results demonstrate that the algorithm is practical—the average time for checking compliance for each disclosure of protected health information is 0.12 seconds for a log of size 15MB.

In practice, we expect that the reduce algorithm will be an integral component of an *after-the-fact audit* tool for policy violations. The other component of such a tool would be a front-end that allows an auditor to select a part of the policy (or the whole of it) and simplify it through the reduce algorithm. The auditor may also provide additional information about incomplete parts of the audit log and repeat the reduction. In ongoing work, we are building such a front-end.

Organization. Section 2 presents the syntax of the policy logic, and defines partial structures and the semantics of the logic over them. Section 3 presents the reduce algorithm and its properties. Section 4 describes the base implementation of the algorithm and its optimization. Section 5 describes our empirical evaluation of the implementation. Section 6 provides a detailed comparison with related work and Section 7 presents conclusions and directions for future work. The source code of our implementation and a technical report [17] with proofs of theorems and other details (TR in the sequel) are available online at the URL `http://www.cs.cmu.edu/~dg/`.

2. POLICY LOGIC

We represent policies in a first-order logic with restricted quantifiers (also called guarded quantifiers in the literature).

Atoms	P	::=	$p(t_1, \ldots, t_n)$
Formulas	φ	::=	$P \mid \top \mid \bot \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid$ $\forall \vec{x}.(c \supset \varphi) \mid \exists \vec{x}.(c \wedge \varphi)$
Restrictions	c	::=	$P \mid \top \mid \bot \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \exists x.c$

Predicates p represent relations between terms t. Terms are variables $(x, y, \ldots)$, constants, or terms applied to uninterpreted function symbols. An atom is a predicate applied to a list of terms. Propositional connectives $\top$ (true), $\bot$ (false), $\wedge$ (conjunction), and $\vee$ (disjunction) have their usual meanings. First order quantifiers — forall ($\forall$) and exists ($\exists$) — may range over infinite domains. Anticipating the requirements of our audit algorithm (Section 3), we restrict these quantifiers to the forms $\forall \vec{x}.(c \supset \varphi)$ and $\exists \vec{x}.(c \wedge \varphi)$ by including a formula c called a *restriction*. By definition, $\forall \vec{x}.(c \supset \varphi)$ is true iff all instances of variables $\vec{x}$ that satisfy c, also satisfy φ. Similarly, $\exists \vec{x}.(c \wedge \varphi)$ holds iff there is an instance of $\vec{x}$ that satisfies both c and φ. To ensure that our enforcement algorithm terminates, we require that only a finite number of substitutions for quantified variables make a restriction true. The latter is forced by the limited syntax of restrictions (note that universal quantifiers and implications are not allowed in restrictions) and other checks that we describe in Section 3.

For technical reasons, we do not include negation in the logic. Instead we assume that each predicate p has a dual $\overline{p}$ such that $p(t_1, \ldots, t_n)$ is true iff $\overline{p}(t_1, \ldots, t_n)$ is false and define a dual $\overline{\varphi}$ that behaves exactly as $\neg\varphi$ would. For example,

$$\begin{array}{rcl} \overline{p(t_1, \ldots, t_n)} & = & \overline{p}(t_1, \ldots, t_n) \\ \overline{\varphi \wedge \psi} & = & \overline{\varphi} \vee \overline{\psi} \\ \overline{\forall \vec{x}.(c \supset \varphi)} & = & \exists \vec{x}.(c \wedge \overline{\varphi}) \\ \overline{\exists \vec{x}.(c \wedge \varphi)} & = & \forall \vec{x}.(c \supset \overline{\varphi}) \end{array}$$

To represent time-dependent (temporal) properties, we assume a domain of integers (both negative and positive integers), denoted τ, that count time in seconds from a fixed point of reference and make the time of occurrence of an event explicit in the predicate that represents the event. For example, the atom $\mathtt{send}(p_1, p_2, m, \tau)$ means that principal p_1 sends to principal p_2 the message m *at time* τ. The relation $\tau_1 \leq \tau_2$ represents the total order on integers. All of linear-time temporal logic (LTL) [24] and its extension TPTL [1] can be encoded in our logic. For details of the encoding see our prior work [15].

Example 2.1. Consider the following policy about transmission of health information from one entity (e.g., a hospital or doctor) to another. This policy is motivated by similar requirements in HIPAA, but is simpler and serves as a good illustration.

> An entity may send an individual's protected health information (phi) to another entity only if the receiving entity is the individual's doctor and the purpose of the transmission is treatment, or the individual has previously consented to the transmission.

To formalize this policy in our logic, we start by assuming that all transmissions made by an entity are recorded in a log. The predicate $\texttt{send}(p_1, p_2, m, \tau)$ is true if the transmission of message m from principal p_1 to principal p_2 at time τ occurs in this log. Similarly, we assume that transmission consents given by individuals are also recorded in a database table. The predicate $\texttt{consents}(q, a, \tau)$, which means that individual q consents to the action a at time τ, holds if the corresponding consent exists in this table.

We further assume that each transmitted message m is *tagged* by the sender (in a machine-readable format) with the names of individuals whose information it carries as well as the attributes of information it carries (attributes include "address", "social security number", "medications", "medical history", etc.). The predicate $\texttt{tagged}(m, q, t)$ means that message m is tagged as carrying individual q's attribute t. Tags may or may not be accurate. Similarly, we assume that each message m is labeled in a machine readable format with a purpose u (e.g., "treatment", "healthcare", etc.). This is represented by the predicate $\texttt{purp}(m, u)$.

Attributes and purposes are assumed to have separate hierarchies, e.g., the attribute "medications" is contained in "medical history". This is formalized as the predicate $\texttt{attr\_in}(\text{medications}, \text{medical-history})$. Similarly, the predicate $\texttt{purp\_in}(u, u')$ means that purpose u is a special case of purpose u', e.g., $\texttt{purp\_in}(\text{surgery}, \text{treatment})$. Finally, $\texttt{doctorOf}(p_2, q, \tau)$ means that q is a doctor of p_2 at time τ.

The policy above can be formalized in our logic as follows (terms *phi* and *treatment* are constants).

$$
\begin{aligned}
&\varphi_{pol} = \\
&\forall p_1, p_2, m, u, q, t, \tau.\ (\texttt{send}(p_1, p_2, m, \tau) \wedge \texttt{purp}(m, u) \wedge \\
&\qquad\qquad \texttt{tagged}(m, q, t)) \\
&\quad \supset \overline{\texttt{attr\_in}}(t, \mathit{phi}) \\
&\qquad \vee (\texttt{doctorOf}(p_2, q, \tau) \wedge \texttt{purp\_in}(u, \mathit{treatment})) \\
&\qquad \vee \exists \tau'.(\tau' < \tau \wedge \\
&\qquad\qquad \texttt{consents}(q, \texttt{sendaction}(p_1, p_2, (q, t)), \tau'))
\end{aligned}
$$

In words, if entity p_1 sends to entity p_2 a message m at time τ, m is tagged as carrying attribute t of individual q, and m is labeled with purpose u, then either the attribute t is not a form of protected health information (so the policy does not apply) or the recipient p_2 is a doctor of q at time τ (atom $\texttt{doctorOf}(p_2, q, \tau)$) and u is a type of treatment, or q has consented to this transmission in the past (last two lines of φ_{pol}).

Predicates may be verified in different ways in an implementation. The predicates `send` and `consents` can be verified by looking up respective logs. Predicates $\texttt{tagged}(m, q, t)$ and $\texttt{purp}(m, u)$ can be verified by examining the tags in m, i.e., through a pre-defined computable function. Similarly, predicates `attr_in` and `purp_in` may be verified through a function that checks stipulated hierarchies over attributes and purposes, respectively. Finally, the predicate $\texttt{doctorOf}(p_2, q, \tau)$ may require human input to resolve because, in general, information about all of q's doctors may be unavailable to the audit mechanism. Our implementation (Section 4) requires the policy designer to categorize predicates based on how they are verified, but our audit algorithm uses a single, abstract representation of these verification methods, called partial structures, to which we turn next.

2.1 Partial Structures and Semantics

We formally abstract the information about truth and falsity of atoms available to our audit algorithm as functions called *partial structures* and define semantics of logical formulas over such structures. Given a possibly infinite domain D of terms, a partial structure (abbrev. structure) $\mathcal{L}$ is a map from atoms over D to the three-value set $\{\texttt{tt}, \texttt{ff}, \texttt{uu}\}$. We say that the atom P is true, false, or unknown in the structure $\mathcal{L}$ if $\mathcal{L}(P)$ is `tt`, `ff`, or `uu`, respectively. The possibility of mapping an atom to "unknown" captures the common phenomena that, during audit, not every atom may be classifiable as true or false. In particular, partial structures abstract the following different kinds of *incompleteness* in information available to the audit algorithm.

- Future incompleteness: Information about events in the future cannot be available to an audit algorithm. For instance, the policy may allow a disclosure if the subject of the information disclosed is notified within a month. However, if an audit occurs immediately after a disclosure, it will not be known whether or not a corresponding notification will be sent in future. This is easily modeled by a partial structure $\mathcal{L}$ satisfying $\mathcal{L}(\texttt{send}(p_1, p_2, m, \tau)) = \texttt{uu}$ for every τ greater than the time of audit.

- Spatial incompleteness: Not all relevant audit logs may be available to the audit system. For instance, with reference to Example 2.1, it is conceivable that the predicates `send` and `consents` are stored on separate physical sites. If we audit at the first site, information about `consents` may be unavailable. This incompleteness is easily modeled by requiring $\mathcal{L}(\texttt{consents}(p, a, \tau)) = \texttt{uu}$ for all p, a and τ.

- Subjective incompleteness: A mechanized audit system is unlikely to resolve predicates that rely on human input. In Example 2.1, assuming that the set of doctors of a patient is not known to the audit algorithm, resolving the predicate $\texttt{doctorOf}(p_2, q, \tau)$ may require human input. Formally, this is modeled by a partial structure $\mathcal{L}$ satisfying $\mathcal{L}(\texttt{doctorOf}(p_2, q, \tau)) = \texttt{uu}$ for all p_2 , q, and τ. Similarly, predicates that rely on human belief or professional judgment, which constitute a significant fraction of all predicates used in a prior formalization of HIPAA [15], can be modeled by mapping them to `uu` in a structure.

Because our audit algorithm (Section 3) works with partial structures, it takes into account all these forms of incompleteness. We note that real audit logs often only list atoms that are true (`tt`), and cannot distinguish atoms that are false (`ff`) from those that are unknown (`uu`). Consequently,

for modeling real scenarios, we define partial structures $\mathcal{L}$ from system logs and additional information about their completeness, as explained in Section 4.1.

Semantics. We formalize the semantics of logical formulas as the relation $\mathcal{L} \models \varphi$, read "$\varphi$ is true in the partial structure $\mathcal{L}$". Restrictions c are a subsyntax of formulas φ, so we do not define the relation separately for them.

- $\mathcal{L} \models P$ iff $\mathcal{L}(P) = \mathtt{tt}$
- $\mathcal{L} \models \top$
- $\mathcal{L} \models \varphi \wedge \psi$ iff $\mathcal{L} \models \varphi$ and $\mathcal{L} \models \psi$
- $\mathcal{L} \models \varphi \vee \psi$ iff $\mathcal{L} \models \varphi$ or $\mathcal{L} \models \psi$
- $\mathcal{L} \models \forall \vec{x}.(c \supset \varphi)$ iff for all $\vec{t} \in \mathrm{D}$ either $\mathcal{L} \models \overline{c}[\vec{t}/\vec{x}]$ or $\mathcal{L} \models \varphi[\vec{t}/\vec{x}]$
- $\mathcal{L} \models \exists \vec{x}.(c \wedge \varphi)$ iff there exists $\vec{t} \in \mathrm{D}$ such that $\mathcal{L} \models c[\vec{t}/\vec{x}]$ and $\mathcal{L} \models \varphi[\vec{t}/\vec{x}]$

For dual atoms, we define $\mathcal{L}(\overline{P}) = \overline{\mathcal{L}(P)}$, where $\overline{\mathtt{tt}} = \mathtt{ff}$, $\overline{\mathtt{ff}} = \mathtt{tt}$, and $\overline{\mathtt{uu}} = \mathtt{uu}$. We say that a formula φ is *false* on the structure $\mathcal{L}$ if $\mathcal{L} \models \overline{\varphi}$. The following two properties hold:

1. Consistency: A formula φ cannot be simultaneously true and false in the structure $\mathcal{L}$, i.e., either $\mathcal{L} \not\models \varphi$ or $\mathcal{L} \not\models \overline{\varphi}$
2. Incompleteness: A formula φ may be neither true nor false in a structure $\mathcal{L}$, i.e., $\mathcal{L} \not\models \varphi$ and $\mathcal{L} \not\models \overline{\varphi}$ may both hold.

The first property follows by induction on φ. The second property follows from a simple example. Consider a structure $\mathcal{L}$ and an atom P such that $\mathcal{L}(P) = \mathtt{uu}$. Then, $\mathcal{L} \not\models P$ and $\mathcal{L} \not\models \overline{P}$.

Structure Extension. In practice, system logs evolve over time by gathering more information. This leads to a partial order, $\mathcal{L}_1 \leq \mathcal{L}_2$ on structures ($\mathcal{L}_2$ *extends* $\mathcal{L}_1$), meaning that $\mathcal{L}_2$ has more information than $\mathcal{L}_1$. Formally, $\mathcal{L}_1 \leq \mathcal{L}_2$ if for all ground atoms P (atoms P without free variables), $\mathcal{L}_1(P) \in \{\mathtt{tt}, \mathtt{ff}\}$ implies $\mathcal{L}_2(P) = \mathcal{L}_1(P)$. Thus, as structures extend, the valuation of an atom may change from $\mathtt{uu}$ to either $\mathtt{tt}$ or $\mathtt{ff}$, but cannot change once it is either $\mathtt{tt}$ or $\mathtt{ff}$. The following property follows by induction on φ:

- Monotonicity: $\mathcal{L}_1 \leq \mathcal{L}_2$ and $\mathcal{L}_1 \models \varphi$ imply $\mathcal{L}_2 \models \varphi$.

Replacing φ with $\overline{\varphi}$, we also obtain that $\mathcal{L}_1 \leq \mathcal{L}_2$ and $\mathcal{L}_1 \models \overline{\varphi}$ imply $\mathcal{L}_2 \models \overline{\varphi}$. Hence, if $\mathcal{L}_1 \leq \mathcal{L}_2$ then $\mathcal{L}_2$ preserves both the $\mathcal{L}_1$-truth and $\mathcal{L}_1$-falsity of every formula φ.

3. AUDIT ALGORITHM

Our main technical contribution is an iterative process that checks for violations of policies written in the logic. At each iteration, our algorithm takes as input a policy φ and information about atoms abstracted as a partial structure $\mathcal{L}$, and outputs a residual policy ψ that contains exactly the parts of φ that could not be verified due to lack of information in $\mathcal{L}$. Such an iteration is written $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$. When more information becomes available, extending $\mathcal{L}$ to $\mathcal{L}'$ ($\mathcal{L} \leq \mathcal{L}'$), another iteration of the algorithm can be used with inputs ψ and $\mathcal{L}'$ to obtain a new formula ψ'. This process can be continued until the output is either $\top$ (no violation), $\bot$ (violation) or a human auditor inspects the output. By design, our algorithm satisfies three important properties:

- Termination: Each iteration terminates.
- Correctness: If $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$, then for all extensions $\mathcal{L}'$ of $\mathcal{L}$, $\mathcal{L}' \models \varphi$ iff $\mathcal{L}' \models \psi$.
- Minimality: If $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$, then an atom occurs in ψ only if it occurs in φ and its valuation on $\mathcal{L}$ is $\mathtt{uu}$.

The technically difficult part of reduce is its treatment of quantifiers over infinite domains. Consider, for instance, the behavior of an algorithm satisfying the above three properties on input $\forall x.\varphi$. Because the output must be minimal, in order to reduce $\forall x.\varphi$, a naive algorithm will instantiate x with each possible element of the domain D and check the truth or falsity of φ for that instance on $\mathcal{L}$. This immediately leads to non-termination if the domain D is infinite, which does happen for real policies (e.g., HIPAA contains quantification over the infinite domain of messages).

Given the need for infinite domains, something intrinsic in quantification must limit the number of *relevant instances* of x that need to be checked to a finite number. To this end, we rely on the restrictions c in quantifiers, $\forall \vec{x}.(c \supset \varphi)$ and $\exists \vec{x}.(c \wedge \varphi)$, and use the technique of *mode analysis* from logic programming [2] to ensure that the restriction c has only a finite number of satisfying instances in any structure and that these instances are *computable*.

Briefly, mode analysis requires the policy designer to specify which argument positions of a predicate can be *computed finitely* from others. For instance, in Example 2.1 we assumed that the purpose of a message is written on it in machine-readable format and, hence, can be computed from the message. Denoting required inputs by $+$ and computable outputs by $-$, we may give the predicate $\mathtt{purp}(m, u)$ the *mode* $\mathtt{purp}(+, -)$, meaning that from the input m, the output u can be computed. The mode $\mathtt{purp}(-, +)$ is incorrect because given a fixed second argument (purpose), there may be an infinite number of first arguments (messages) annotated with that purpose, so the latter set cannot be finitely computed. Similarly, if the predicate $\mathtt{mult}(x, y, z)$ means that $x = yz$, where x, y, z are integers, then any of the modes $\mathtt{mult}(+, +, -)$, $\mathtt{mult}(-, +, +)$, and $\mathtt{mult}(+, -, +)$ are okay, but $\mathtt{mult}(-, -, +)$ is not.

Given the mode information of all predicates in a policy, a static, linear-time check of the policy, called a *mode check*, ensures that there are only a finite number of instances of free variables that can satisfy a restriction c in the policy. To keep the presentation accessible, we omit a technical description of mode checking, but its details are present in our TR. We note that mode checking is very permissive even though not every policy in the syntax of the logic passes the mode check. In particular, our entire prior formalization of HIPAA [15] passes the check.

To actually *compute* the satisfying instances of a restriction, we define a function $\widehat{\mathtt{sat}}(\mathcal{L}, c)$ that returns all substitutions σ for free variables of c such that $\mathcal{L} \models c\sigma$ (Section 3.1). This definition assumes a function $\mathtt{sat}(\mathcal{L}, P)$ that returns all substitutions σ for free variables of P such that $\mathcal{L} \models P\sigma$ if all input positions in P are ground. In practice, $\mathtt{sat}(\mathcal{L}, P)$ is implemented by looking up system logs or calling domain-specific solvers, depending on the predicate in

P (see Section 4 for the definition of $\texttt{sat}$ that we use in our implementation).

Finally, the main audit function $\mathsf{reduce}(\mathcal{L}, \varphi)$ is defined by induction on φ, using $\widehat{\texttt{sat}}(\mathcal{L}, c)$ as a black-box. Because $\texttt{sat}(\mathcal{L}, P)$ may only be defined for P with ground input arguments, $\mathsf{reduce}(\mathcal{L}, \varphi)$ is a partial function. However, we show that if φ passes the mode check, then $\mathsf{reduce}(\mathcal{L}, \varphi)$ is defined (Theorem 3.3).

3.1 Iterative Reduction

At the core of our audit regime is a computable function $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$, that instantiates quantifiers in, and simplifies, the prevalent policy φ using information from the extant structure $\mathcal{L}$ to obtain a residual policy ψ. Given an initial policy φ_0 and a sequence of structures $\mathcal{L}_1 \leq \mathcal{L}_2 \leq \ldots \leq \mathcal{L}_n$, the reduction algorithm can be applied repeatedly to obtain $\varphi_1, \ldots, \varphi_n$ such that $\mathsf{reduce}(\mathcal{L}_i, \varphi_{i-1}) = \varphi_i$. We write this process in symbols as $\varphi_0 \xrightarrow{\mathcal{L}_1} \varphi_1 \ldots \xrightarrow{\mathcal{L}_n} \varphi_n$.

The definition of reduce is shown in Figure 1. It relies on the function $\widehat{\texttt{sat}}(\mathcal{L}, c)$, which we define later. $\widehat{\texttt{sat}}(\mathcal{L}, c)$ computes the finite set of substitutions σ such that $\mathcal{L} \models c\sigma$. For atoms P, $\mathsf{reduce}(\mathcal{L}, P)$ equals $\top$, $\bot$, or P, if $\mathcal{L}(P)$ equals $\texttt{tt}$, $\texttt{ff}$, or $\texttt{uu}$, respectively. The clauses for the connectives $\top$, $\bot$, $\wedge$, and $\vee$ are straightforward. To evaluate $\mathsf{reduce}(\mathcal{L}, \forall\vec{x}.(c \supset \varphi))$, we first determine the set of instances of $\vec{x}$ that satisfy c by calling $\widehat{\texttt{sat}}(\mathcal{L}, c)$. For each such instance $\vec{t_1}, \ldots, \vec{t_n}$, we reduce $\varphi[\vec{t_i}/\vec{x}]$ to ψ_i through a recursive call to reduce. Because all instances of φ must hold in order for $\forall\vec{x}.(c \supset \varphi)$ to be true, the output is $\psi_1 \wedge \ldots \wedge \psi_n \wedge \psi'$, where the last conjunct ψ' records the fact that instances of $\vec{x}$ *other than* $\vec{t_1}, \ldots, \vec{t_n}$ have not been considered. The latter is necessary because there may be instances of $\vec{x}$ satisfying c in extensions of $\mathcal{L}$, but not $\mathcal{L}$ itself. Precisely, we define $S = \{\vec{t_1}, \ldots, \vec{t_n}\}$ and $\psi' = \forall\vec{x}.((c \wedge \vec{x} \notin S) \supset \varphi)$. The new conjunct $\vec{x} \notin S$ prevents the instances $\vec{t_1}, \ldots, \vec{t_n}$ from being checked again in subsequent iterations. Formally, $\vec{x} \notin S$ encodes the negation of the usual finite-set membership. The treatment of $\exists\vec{x}.(c \wedge \varphi)$ is dual; in that case, the output contains disjunctions because the truth of any one instance of φ suffices for the formula to hold.

Our implementation also performs trivial rewriting to simplify the output of reduce. Specifically, it rewrites $\psi \wedge \top$ to ψ, $\psi \wedge \bot$ to $\bot$, $\psi \vee \top$ to $\top$, and $\psi \vee \bot$ to ψ.

Definition of $\widehat{\texttt{sat}}$. The function $\widehat{\texttt{sat}}(\mathcal{L}, c)$ which computes the set of substitutions σ such that $\mathcal{L} \models c\sigma$, is defined below. This function relies on a given function $\texttt{sat}(\mathcal{L}, P)$ that computes the set of substitutions σ such that $\mathcal{L} \models P\sigma$. The latter function is application-dependent, as described in Section 4.

$$
\begin{array}{lcl}
\widehat{\texttt{sat}}(\mathcal{L}, p(t_1, \ldots, t_n)) & = & \texttt{sat}(\mathcal{L}, p(t_1, \ldots, t_n)) \\
\widehat{\texttt{sat}}(\mathcal{L}, \top) & = & \{\bullet\} \\
\widehat{\texttt{sat}}(\mathcal{L}, \bot) & = & \{\} \\
\widehat{\texttt{sat}}(\mathcal{L}, c_1 \wedge c_2) & = & \bigcup_{\sigma \in \widehat{\texttt{sat}}(\mathcal{L}, c_1)} \sigma + \widehat{\texttt{sat}}(\mathcal{L}, c_2\sigma) \\
\widehat{\texttt{sat}}(\mathcal{L}, c_1 \vee c_2) & = & \widehat{\texttt{sat}}(\mathcal{L}, c_1) \cup \widehat{\texttt{sat}}(\mathcal{L}, c_2) \\
\widehat{\texttt{sat}}(\mathcal{L}, \exists x.c) & = & \widehat{\texttt{sat}}(\mathcal{L}, c) \backslash \{x\} \qquad (x \text{ fresh})
\end{array}
$$

For atoms, the definition of $\widehat{\texttt{sat}}$ coincides with that of $\texttt{sat}$. Since $\top$ must always be true, $\widehat{\texttt{sat}}(\mathcal{L}, \top)$ contains only the empty substitution (denoted $\bullet$). Since $\bot$ can never be satisfied, $\widehat{\texttt{sat}}(\mathcal{L}, \bot)$ is empty (denoted $\{\}$). For $c_1 \wedge c_2$, the set of satisfying instances is obtained by taking those of c_1 (denoted σ above), and conjoining those with satisfying instances of $c_2\sigma$ (the operation $\sigma + \Sigma$ appends σ to every substitution in Σ). The set of satisfying instances of $c_1 \vee c_2$ is the union of the satisfying instances of c_1 and c_2. Satisfying instances of $\exists x.c$ are obtained by taking those of c, and removing the substitutions for x ($\Sigma\backslash\{x\}$ removes x from the domain of every substitution in Σ).

$$
\begin{array}{lcl}
\mathsf{reduce}(\mathcal{L}, P) & = & \begin{cases} \top & \text{if } \mathcal{L}(P) = \texttt{tt} \\ \bot & \text{if } \mathcal{L}(P) = \texttt{ff} \\ P & \text{if } \mathcal{L}(P) = \texttt{uu} \end{cases} \\
\mathsf{reduce}(\mathcal{L}, \top) & = & \top \\
\mathsf{reduce}(\mathcal{L}, \bot) & = & \bot \\
\mathsf{reduce}(\mathcal{L}, \varphi_1 \wedge \varphi_2) & = & \mathsf{reduce}(\mathcal{L}, \varphi_1) \wedge \mathsf{reduce}(\mathcal{L}, \varphi_2) \\
\mathsf{reduce}(\mathcal{L}, \varphi_1 \vee \varphi_2) & = & \mathsf{reduce}(\mathcal{L}, \varphi_1) \vee \mathsf{reduce}(\mathcal{L}, \varphi_2) \\
\mathsf{reduce}(\mathcal{L}, \forall\vec{x}.(c \supset \varphi)) & = & \text{let} \\
 & & \quad \{\sigma_1, \ldots, \sigma_n\} \leftarrow \widehat{\texttt{sat}}(\mathcal{L}, c) \\
 & & \quad \{\vec{t_i} \leftarrow \sigma_i(\vec{x})\}_{i=1}^n \\
 & & \quad S \leftarrow \{\vec{t_1}, \ldots, \vec{t_n}\} \\
 & & \quad \{\psi_i \leftarrow \mathsf{reduce}(\mathcal{L}, \varphi[\vec{t_i}/\vec{x}])\}_{i=1}^n \\
 & & \quad \psi' \leftarrow \forall\vec{x}.((c \wedge \vec{x} \notin S) \supset \varphi) \\
 & & \text{return} \\
 & & \quad \psi_1 \wedge \ldots \wedge \psi_n \wedge \psi' \\
\mathsf{reduce}(\mathcal{L}, \exists\vec{x}.(c \wedge \varphi)) & = & \text{let} \\
 & & \quad \{\sigma_1, \ldots, \sigma_n\} \leftarrow \widehat{\texttt{sat}}(\mathcal{L}, c) \\
 & & \quad \{\vec{t_i} \leftarrow \sigma_i(\vec{x})\}_{i=1}^n \\
 & & \quad S \leftarrow \{\vec{t_1}, \ldots, \vec{t_n}\} \\
 & & \quad \{\psi_i \leftarrow \mathsf{reduce}(\mathcal{L}, \varphi[\vec{t_i}/\vec{x}])\}_{i=1}^n \\
 & & \quad \psi' \leftarrow \exists\vec{x}.((c \wedge \vec{x} \notin S) \wedge \varphi) \\
 & & \text{return} \\
 & & \quad \psi_1 \vee \ldots \vee \psi_n \vee \psi'
\end{array}
$$

Figure 1: Definition of $\mathsf{reduce}(\mathcal{L}, \varphi)$

Example 3.1. We illustrate iterative audit on the policy φ_{pol} from Example 2.1. For notational convenience, let $\vec{x}$ denote the sequence of variables $p_1, p_2, m, u, q, t, \tau$ and define $c(\vec{x})$ and $\varphi(\vec{x})$ by pattern matching as the restriction and formula satisfying $\varphi_{pol} = \forall\vec{x}.\ c(\vec{x}) \supset \varphi(\vec{x})$. Intuitively, $\varphi(p_1, p_2, m, u, q, t, \tau)$ is the formula that must be satisfied if p_1 sends to p_2 the message m at time τ and m is tagged as containing attribute t about principal q for purpose u. Further, define φ_2 and φ_3 by pattern matching as follows: $\varphi(\vec{x}) = \overline{\texttt{attr\_in}}(t, phi) \vee \varphi_2(p_2, q, \tau, u) \vee \varphi_3(\tau, q, p_1, p_2, t)$.

Consider a structure $\mathcal{L}$ with the following information: (1) Alice sends to Bob the message M at time 4, tagged as containing information about Charlie's address for the purpose of billing, (2) Charlie authorized this transmission at time 2. This information implies that $\widehat{\texttt{sat}}(\mathcal{L}, c(\vec{x})) = \{\sigma\}$, where $\sigma = [\vec{x} \mapsto (\text{Alice}, \text{Bob}, M, \text{billing}, \text{Charlie}, \text{address}, 4)]$. Applying the definition of reduce, we obtain $\mathsf{reduce}(\mathcal{L}, \varphi_{pol}) = \psi_1 \wedge \varphi'_{pol}$ where $\psi_1 = \mathsf{reduce}(\mathcal{L}, \varphi(\vec{x})\sigma)$ and $\varphi'_{pol} = \forall\vec{x}.\ (c(\vec{x}) \wedge \vec{x} \notin \{\sigma\}) \supset \varphi(\vec{x})$.

Since $\varphi(\vec{x})$ is a disjunction of three formulas, the third of which is $\varphi_3(\tau, q, p_1, p_2, t)$, ψ_1 is also a disjunction of three formulas, the third of which is $\mathsf{reduce}(\mathcal{L}, \varphi_3(\tau, q, p_1, p_2, t)\sigma)$. It can easily be shown using (2) that this third disjunct is $\top$, so ψ_1 also simplifies to $\top$. This indicates that there is no vi-

olation of the policy so far, which should be intuitively clear from the description of the structure $\mathcal{L}$ and an inspection of the policy φ_{pol}. Succinctly, we have $\mathsf{reduce}(\mathcal{L}, \varphi_{pol}) = \varphi'_{pol}$.

Next, consider an extension $\mathcal{L}'$ which adds new information: (3) Alice sends to Bob a message M' at time 5, tagged as containing Dan's lab report for the purpose of surgery, (4) $\mathcal{L}'(\texttt{consents}(\text{Dan}, a, \tau')) = \texttt{ff}$ for every action a and every time τ' (in particular, Dan has not consented to Alice's transmission), and (5) $\mathcal{L}'(\texttt{doctorOf}(\text{Bob}, \text{Dan}, 5)) = \texttt{ff}$ (so Bob is not Dan's doctor at time 5). Then, the restriction in the top-level universal quantifier of φ'_{pol}, i.e., $c'(\vec{x}) = c(\vec{x}) \wedge \vec{x} \notin \{\sigma\}$ satisfies $\widehat{\texttt{sat}}(\mathcal{L}', c'(\vec{x})) = \{\sigma'\}$, where $\sigma' = [\vec{x} \mapsto (\text{Alice}, \text{Bob}, M', \text{surgery}, \text{Dan}, \text{labreport}, 5)]$. Hence, $\mathsf{reduce}(\mathcal{L}', \varphi'_{pol}) = \psi_2 \wedge \varphi''_{pol}$, where $\psi_2 = \mathsf{reduce}(\mathcal{L}', \varphi(\vec{x})\sigma')$ and φ''_{pol} enforces the policy on transmissions other than the two already considered. Calculation using (4) and (5) yields $\psi_2 = \overline{\texttt{attr\_in}}(\text{labreport}, \mathit{phi})$, meaning that Alice's transmission of Bob's record satisfies the policy only if labreport is not a form of *phi* (protected health information). The same conclusion may also be expected from an informal inspection of the policy.

Reporting Causes of Policy Violations. From what we have presented so far, it may appear that the output of $\mathsf{reduce}(\mathcal{L}, \varphi)$ can only say that there has been a policy violation but cannot indicate *which instances* of quantifiers in φ have led to the violation. For instance, consider the policy $\varphi_{pol} = \forall \vec{x}.\ (c(\vec{x}) \supset \varphi(\vec{x}))$ defined in the beginning of Example 3.1. If $c(\vec{a})$ is true for some instance $\vec{a}$ of $\vec{x}$ on structure $\mathcal{L}$, but $\varphi(\vec{a})$ is false, then $\mathsf{reduce}(\mathcal{L}, \varphi_{pol})$ will evaluate to $\bot$, indicating a violation of the policy. However, this output does not indicate the instance $\vec{a}$ which actually causes the violation.

Violating instances of a policy can be reported in the output of reduce by slightly rewriting the policy formula (but without changing the the reduce algorithm). The idea is straightforward and easily illustrated with the example of φ_{pol}. Consider a new predicate $\texttt{violated}(\vec{x})$ which means that the list of terms $\vec{x}$ causes a violation of the policy φ_{pol}. Further, and importantly, assume that violated is uninterpreted, i.e., for all $\vec{t}$, $\mathcal{L}(\texttt{violated}(\vec{t})) = \texttt{uu}$ (in our implementation, we call such uninterpreted predicates subjective or SUBJ; see Section 4). Consider the revised policy $\varphi'_{pol} = \forall \vec{x}.(c(\vec{x}) \supset (\texttt{violated}(\vec{x}) \vee \varphi(\vec{x})))$. Intuitively, φ'_{pol} says that for all instances $\vec{a}$ of $\vec{x}$, if $c(\vec{a})$ is true, then either $\vec{a}$ violates the policy or $\varphi(\vec{a})$ is true. This is also the intended meaning of φ_{pol}, so replacing it with φ'_{pol} does not change the meaning of the policy itself. However, when reduce is executed on φ'_{pol} instead of φ_{pol}, the top-level of the output contains $\texttt{violated}(\vec{a_1}) \wedge \ldots \wedge \texttt{violated}(\vec{a_n})$, where $\vec{a_1}, \ldots, \vec{a_n}$ are exactly those instances of $\vec{x}$ for which c is true and φ is false. So, all instances of the quantifier $\vec{x}$ that cause policy violation can be read from the output of reduce on the revised policy φ'_{pol}. The technique illustrated by this example generalizes easily; any quantified formula (either universal or existential) can be modified by adding a new uninterpreted predicate to cause reduce to report all instances of quantified variables that violate the formula.

3.2 Properties of the Audit Algorithm

Partial Correctness. The function reduce is partially correct: If $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$, then in all extensions of $\mathcal{L}$, φ and ψ are logically equivalent.

Theorem 3.2 (Partial correctness of reduce). *If* $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$ *and* $\mathcal{L} \leq \mathcal{L}'$, *then (1)* $\mathcal{L}' \models \varphi$ *iff* $\mathcal{L}' \models \psi$ *and (2)* $\mathcal{L}' \models \overline{\varphi}$ *iff* $\mathcal{L}' \models \overline{\psi}$.

Partial correctness of iterative audit is an immediate corollary of Theorem 3.2. We can prove by induction on n that if $\varphi_0 \xrightarrow{\mathcal{L}_1} \varphi_1 \ldots \xrightarrow{\mathcal{L}_n} \varphi_n$, then for all structures $\mathcal{L}'$ satisfying $\mathcal{L}_n \leq \mathcal{L}'$, $\mathcal{L}' \models \varphi_n$ iff $\mathcal{L}' \models \varphi_0$ and $\mathcal{L}' \models \overline{\varphi_n}$ iff $\mathcal{L}' \models \overline{\varphi_0}$.

Totality. Let $\vdash \varphi$ mean that φ passes the mode check. The following theorem states that reduce is defined on every such φ and, further, that the output also passes the mode check.

Theorem 3.3 (Totality of reduce). *If* $\vdash \varphi$ *then there is a* ψ *such that* $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$ *and* $\vdash \psi$.

Complexity. The function $\mathsf{reduce}(\mathcal{L}, \varphi)$ can be implemented using auxiliary space polynomial in the size of the formula φ, and in time polynomial in the maximum number of substitutions returned by sat (denoted $|\mathcal{L}|$) where the degree of the polynomial is proportional to the size of the formula φ. This analysis assumes that computation of each output returned by sat takes unit time which, in turn means that finding a single row in a system log and reading it takes unit time. In practice, $|\mathcal{L}|$ is bounded by the size of the system logs, so the execution time of reduce for a fixed policy is polynomial in the size of the system logs.

Theorem 3.4 (Complexity of reduce). *Assuming that computing each output returned by* sat *takes unit time, the algorithm* $\mathsf{reduce}(\mathcal{L}, \varphi)$ *lies in the intersection of the complexity classes TIME*$(|\mathcal{L}|^{O(|\varphi|)})$ *and PSPACE*$(|\varphi|)$.

Minimality. If $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$ then ψ is minimal with respect to φ and $\mathcal{L}$, i.e., an atom occurs in ψ only if it occurs in φ and its valuation in $\mathcal{L}$ is unknown. Unfortunately, owing to quantification, there is no standard definition of the set of atoms of a formula of first-order logic. In the following, we provide one natural definition of the atoms of a formula and characterize minimality with respect to it. For a formula φ that passes the mode check, we define the set of atoms of φ with respect to a structure $\mathcal{L}$ as follows.

$$
\begin{array}{lcl}
\texttt{atoms}(\mathcal{L}, P) & = & \{P\} \\
\texttt{atoms}(\mathcal{L}, \top) & = & \{\} \\
\texttt{atoms}(\mathcal{L}, \bot) & = & \{\} \\
\texttt{atoms}(\mathcal{L}, \varphi_1 \wedge \varphi_2) & = & \texttt{atoms}(\mathcal{L}, \varphi_1) \cup \texttt{atoms}(\mathcal{L}, \varphi_2) \\
\texttt{atoms}(\mathcal{L}, \varphi_1 \vee \varphi_2) & = & \texttt{atoms}(\mathcal{L}, \varphi_1) \cup \texttt{atoms}(\mathcal{L}, \varphi_2) \\
\texttt{atoms}(\mathcal{L}, \forall \vec{x}.(c \supset \varphi)) & = & \bigcup_{\sigma \in \widehat{\texttt{sat}}(\mathcal{L}, c)} \texttt{atoms}(\mathcal{L}, \varphi\sigma) \\
\texttt{atoms}(\mathcal{L}, \exists \vec{x}.(c \wedge \varphi)) & = & \bigcup_{\sigma \in \widehat{\texttt{sat}}(\mathcal{L}, c)} \texttt{atoms}(\mathcal{L}, \varphi\sigma)
\end{array}
$$

Theorem 3.5 (Minimality). *If* $\vdash \varphi$ *and* $\mathsf{reduce}(\mathcal{L}, \varphi) = \psi$, *then* $\texttt{atoms}(\mathcal{L}, \psi) \subseteq \texttt{atoms}(\mathcal{L}, \varphi) \cap \{P \mid \mathcal{L}(P) = \texttt{uu}\}$.

4. IMPLEMENTATION

We have implemented the iterative algorithm $\mathsf{reduce}(\mathcal{L}, \varphi)$ described in Section 3 as well as the mode analysis on formulas. The implementation is written in Standard ML. This section describes the implementation of $\mathsf{reduce}(\mathcal{L}, \varphi)$ and syntax-directed optimizations on it. Section 5 reports on the implementation's experimental evaluation.

4.1 Application-Specific Structures

Our implementation of reduce closely follows its abstract definition from Section 3.1. The implementation must be *instantiated* for a specific application by providing enough information to define a structure $\mathcal{L}$, specifically, the two functions $\texttt{sat}(\mathcal{L}, P)$ and $\mathcal{L}(P)$, on which reduce relies. This information includes audit logs (e.g., transmission or access logs) and relevant system databases (e.g., the roles database), and is structured into five different components that we describe next. Later, we describe how these four components are used to define $\texttt{sat}(\mathcal{L}, P)$ and $\mathcal{L}(P)$.

First, the implementation of reduce must be told how each predicate is verified. This is called the *category* of the predicate, and is provided to the implementation through a file in a custom syntax. Our current implementation supports the following three common categories, but can be extended modularly to include other forms of verification if needed.

A. DB (Database): The predicate maps directly to a table in a relational database and is computed by looking up the table. The name of the table and the column name corresponding to each argument of the predicate must also be provided. We assume that audit logs are also represented as database tables, so predicates corresponding to these logs (e.g., `send` from Example 2.1) are also in the category DB.

B. EVAL (Evaluable): The predicate is verified through a given computable function (e.g., $\texttt{mult}(x, y, z)$ meaning $x = yz$ would lie in this category).

C. SUBJ (Subjective): The predicate has no mechanized definition; it is checked by a human auditor.

To illustrate, the predicates `send` and `consents` in Example 2.1 are DB because they correspond to tables in the system logs; predicates `tagged` and `purp` are EVAL because they are implemented by analysis of messages (their first arguments); predicate `doctorOf` is SUBJ because its verification requires knowledge of all of a principal's doctors, which is unlikely to be available in mechanized form.

Second, a function, lookup, to query the database must be provided. Our current implementation requires that lookup support standard SQL SELECT queries. Using this function, we define a function $\mathsf{sat\_db}(P)$ that finds all satisfying substitutions for variables in a DB atom P by running a query on the database table corresponding to P.

Third, a function $\mathsf{eval}(P)$ to evaluate EVAL atoms P must be provided. This function has the same specification as `sat`, except that it need be defined only for EVAL atoms.

Fourth, we require a function $\mathsf{isFinal}(P)$ that takes as input a DB or EVAL atom P with ground input arguments and returns true iff in future extensions of the structure, no additional substitutions satisfying P will be found (i.e., substitutions other than those found by looking up its corresponding table or by running eval on it). The implementation of $\mathsf{isFinal}(P)$ will vary based on the predicate in P, but a simple heuristic works for DB atoms P if the database table corresponding to P is available in its entirety at the time of audit: $\mathsf{isFinal}(P) = \text{true}$ iff the time argument of P (which specifies the time at which the truth of P is being ascertained) is less than the time at which the database table was last updated.

Fifth, we require two tables of ground SUBJ atoms subjTrue and subjFalse that contain SUBJ atoms that have been marked true and false respectively by human auditors earlier. These tables are initially empty and an interactive front-end could allow a human auditor to add entries to both tables, making sure that no atom appears in both tables simultaneously.

Given these five components defining a structure $\mathcal{L}$, `sat` can be defined on $\mathcal{L}$ as follows.

$$\texttt{sat}(\mathcal{L}, P) = \begin{cases} \mathsf{sat\_db}(P) & \text{if category}(P) = \text{DB} \\ \mathsf{eval}(P) & \text{if category}(P) = \text{EVAL} \\ \text{undefined} & \text{if category}(P) = \text{SUBJ} \end{cases}$$

(Our mode check ensures that a SUBJ atom never appears in a restriction so, following the definition of reduce, $\texttt{sat}(\mathcal{L}, P)$ is never called if P is SUBJ. Hence, it is okay to have $\texttt{sat}(\mathcal{L}, P) = \text{undefined}$ when P is SUBJ.)

Finally, we define $\mathcal{L}(P)$. Note that $\mathcal{L}(P)$ is meaningful only for P without free variables, and for such P, $\texttt{sat}(\mathcal{L}, P)$ must be either $\{\}$ (no satisfying substitution) or $\{\bullet\}$ (only the trivial satisfying substitution) or undefined.

$$\mathcal{L}(P) = \begin{cases} \texttt{tt} & \text{if } \texttt{sat}(\mathcal{L}, P) = \{\bullet\} \text{ or } P \in \mathsf{subjTrue} \\ \texttt{ff} & \text{if } (\texttt{sat}(\mathcal{L}, P) = \{\} \text{ and } \mathsf{isFinal}(P) = \text{true}) \\ & \quad \text{or } P \in \mathsf{subjFalse} \\ \texttt{uu} & \text{otherwise} \end{cases}$$

The valuation of a variable-free atom P is `tt` if either it has a satisfying substitution or it exists in the table subjTrue; the valuation is `ff` if either P does not have a satisfying substitution and it cannot become true in any extension of the structure or it exists in the table subjFalse; in all other cases, the valuation of P is `uu`.

4.2 Optimizations

We discuss some optimizations to simplify the output of reduce and improve the performance of its implementation.

Removing Residual Terms. The residual formula $\psi' = \forall\vec{x}.((c \wedge \vec{x} \notin S) \supset \varphi)$ or $\psi' = \exists\vec{x}.((c \wedge \vec{x} \notin S) \wedge \varphi)$ in the output of reduce in Figure 1 can be very large because it contains all substitutions S in the output of $\widehat{\texttt{sat}}(\mathcal{L}, c)$. Accordingly, we remove this residual formula from the output of reduce using the following simple heuristic: If during the evaluation of $\widehat{\texttt{sat}}(\mathcal{L}, c)$ in the reduction of a quantifier $\forall\vec{x}.(c \supset \varphi)$ or $\exists\vec{x}.(c \wedge \varphi)$, every recursive call of the form $\texttt{sat}(\mathcal{L}, P)$ satisfies $\mathsf{isFinal}(P) = \text{true}$, then the $c \wedge \vec{x} \notin S$ cannot be satisfied in any extension of $\mathcal{L}$, so the residual term ψ' can be omitted. Whether or not $\mathsf{isFinal}(P) = \text{true}$ in every recursive call is easily checked by maintaining a boolean variable during evaluation of $\widehat{\texttt{sat}}$.

Database Indexing. As may be expected, indexing tables in system databases has a significant impact on performance. We list two reasonable syntax-directed heuristics for choosing tables and columns to index. These heuristics are justified through experiments in Section 5.

- Index deeper: More performance gains may be expected by indexing tables corresponding to DB predicates that are nested deeper inside quantifiers in the policy because our reduce procedure replicates formulas for each instance of a quantifier and this has a multiplicative effect for nested quantifiers.

- Index input modes: A table corresponding to the DB predicate p should be indexed on columns that correspond to input arguments in the mode of p because

these columns are guaranteed to be known in any look-up query on the table.

Database Caching. We mention that in-memory caching of previously read entries of a database table is unlikely to improve performance when policies contain quantifiers. This is because the atoms checked against a database change every time quantifiers are instantiated differently. On the other hand, in-memory caching of database entries is very beneficial for policies without quantifiers.

5. EVALUATION

We evaluate our implementation of the algorithm reduce on policies that regulate the transmission of protected health information in the HIPAA Privacy Rule using synthetic logs generated by a simulator. Section 5.1 explains the setup of our experiments and Section 5.2 reports measurements of running time and memory consumed.

5.1 System Setup

HIPAA Policies. In prior work, we formalized all 84 transmission related clauses of HIPAA in a first-order logic PrivacyLFP [15]. To use our audit algorithm reduce on the formalized policy, we made semantics-preserving changes to our earlier encoding to make it fit our restricted quantifier syntax and to make it pass our mode check. These changes are minor and mostly involve refactoring parts of formulas under quantifiers into the restrictions required by our reduced syntax.

We show below the top-level formula of the HIPAA policy, φ_{HIPAA}. Predicates in φ_{HIPAA} have the same meaning as predicates of the same names in Example 2.1. The new predicate $\texttt{contains}(m, q, t, u)$ means that the disclosed message m is both *correctly* tagged with the subject q and the attribute t of the information it is carrying and is correctly labeled with the purpose of the disclosure u. Verification of correctness of tags requires manual analysis of the content of m, which may be free text and, therefore, the predicate contains has category SUBJ. The body of the HIPAA policy is a conjunction of three components: 1) The contains predicate, 2) a disjunction of the so-called positive norms φ_i^+, of which at least one must be satisfied for every disclosure, and 3) a conjunction of the so-called negative norms φ_i^-, all of which must be satisfied for every disclosure [15].

$$\begin{aligned}&\varphi_{\text{HIPAA}} = \\ &\quad \forall p_1,p_2,m,u,\tau.\ (\texttt{send}(p_1,p_2,m,\tau) \wedge \texttt{purp}(m,u)) \\ &\qquad \supset \forall q,t.\ \texttt{tagged}(m,q,t) \\ &\qquad\quad \supset (\texttt{contains}(m,q,t,u) \wedge \textstyle\bigvee_i \varphi_i^+ \wedge \bigwedge_i \varphi_i^-)\end{aligned}$$

Synthesizing Audit Logs. We test the algorithm reduce on synthetic logs generated by a discrete event-driven simulation that considers several disclosure scenarios governed by HIPAA. In particular, we simulate disclosures of protected health information by a HIPAA covered entity in the following scenarios: For its own treatment, payment and health operations; for another health provider's treatment, payment and health operations; for law-enforcement activities; for judicial administrative proceedings; for notification to friends and family to assist in treatment; and for marketing purposes.

Generated logs contain different types of events (disclosures, role changes, etc.). Each type of event is represented in the logic by a predicate (e.g., disclosures correspond to the predicate send) and is stored in a separate table in a SQLite database.

More precisely, the simulator implements a probabilistic event scheduler. Each scenario is repeated after a probabilistic gap, whose average value is an input to the simulator. For instance, we may assume that, on average, a hospital releases some patient's protected health information for judicial administrative proceedings every 30 days. During the execution of a scenario, the simulator adds relevant events to the database. In some cases, events may not be added with certain probabilities to model policy violations. For instance, one of our scenarios discloses protected health information to a coroner for the purpose of determining the cause of death of a patient. During this scenario, an entry corresponding to the disclosure is created in the database. In addition, two other facts are added to the database, but with probabilities less than 1: The first fact asserts that the receiver is actually in the role of a coroner and the second fact asserts that the purpose of the disclosure is determining the cause of death. Failure to add either of these two facts models a policy violation.

During simulation, we tag each disclosed message with a fresh symbolic attribute, and create entries in a table attr_in_db to record sub-attribute relations between such symbolic attributes. An entry (t_1, t_2) in table attr_in_db means that attribute t_1 is a sub-attribute of t_2. The table attr_in_db is used to compute the EVAL predicate attr_in.

5.2 Performance Evaluation

We perform two sets of experiments: one evaluates a single run of reduce, and the other evaluates iterative runs of reduce in the sense described at the beginning of Section 3.1. All experiments were performed on a 3GHz Intel Core 2 Duo CPU running Linux with 8GB RAM and a 7200 RPM hard disk drive. The database used for storing logs is SQLite version 3.7.2. Our implementation was compiled using the MLton compiler for Standard ML.

In the first set of experiments, we evaluate the impact of database indexing, the size of the database and the rate of policy violations on running time and memory consumption of reduce, by executing it on logs generated by the simulator. We vary the size of, and indexes on, three tables: send, tagged, and attr_in_db. These tables correspond to predicates that are nested progressively deeper inside quantifiers in φ_{HIPAA}. For each experiment, we record the size of the entire log (actually the size of the database containing all logs and indexes on them) and for each table we record the number of entries in the table, and whether or not it is indexed.

Figure 2 summarizes the evaluation results. For instance, in Experiment 1, the size of the entire database is 2.68MB, the table send contains 5401 entries, and it is not indexed (symbols N in the last four columns). Following the heuristic "index input modes" from Section 4.2, whenever we index a database table (entry Y in the last four columns), the index is created only on columns that correspond to input arguments of the corresponding predicate. The fifth column of Figure 2 is the probability that a disclosure in any scenario violates the policy.

Experiments 1–5 demonstrate the effect of indexing the database tables. Indexing a database table allows efficient random read access to data. As mentioned in Section 4.2,

Exp No.	Ave. Time per disclosure (s)	Total time (s)	Memory used (KB)	Prob. of violation	Log size (MB)	Number of entries/indexed (Y or N)			
						`send`	`tagged`	`attr_in_db`	the rest
1	0.27	1453.12	592440	0.10	2.68	5401/N	4947/N	5100/N	-/N
2	0.27	1456.43	592536	0.10	2.99	5401/Y	4947/N	5100/N	-/N
3	0.27	1448.53	592604	0.10	3.14	5401/Y	4947/Y	5100/N	-/N
4	0.26	1378.62	657436	0.10	3.29	5401/Y	4947/Y	5100/Y	-/N
5	0.04	209.21	662080	0.10	3.98	5401/Y	4947/Y	5100/Y	-/Y
6	0.07	725.19	1336808	0.10	7.68	10866/Y	9040/Y	9314/Y	-/Y
7	0.12	2674.77	2375564	0	15.11	21684/Y	16945/Y	17565/Y	-/Y
8	0.12	2680.25	2667132	0.10	15.10	21742/Y	17182/Y	17732/Y	-/Y
9	0.11	2306.93	3565964	1	14.41	21626/Y	16865/Y	17465/Y	-/Y

Figure 2: Experimental evaluation of single runs of the algorithm reduce

we may expect that indexing of predicates that are nested deeper in the policy would improve performance more. The results of Experiments 1–5 verify this expectation. For example, indexing the table of the predicate `send` (Experiment 2), which appears only in the restriction of the top-level quantifier of φ_{HIPAA} does not improve the running time at all over the baseline without any indexing (Experiment 1) because during the execution of reduce on φ_{HIPAA}, the `send` table is not accessed randomly. Instead, it is scanned linearly to instantiate the top-level quantifier in φ_{HIPAA}. Adding an index on the table `tagged` (Experiment 3), which is nested under two quantifiers, improves performance slightly over indexing `send` alone. Indexing `attr_in_db` (Experiment 4), which is nested very deep under quantifiers in the policy results in a noticeable improvement in performance. When we index all other tables in our database, of which six correspond to predicates that are nested very deep under quantifiers in the policy formula, we notice a large improvement in performance (Experiment 5).

Experiments 5, 6 and 8 measure the impact of log size on audit time for fully indexed tables. As expected, both the overall audit time and the per-disclosure audit time increase as the size of the logs increases.

Experiments 7–9 measure the effect of changing the ratio of policy violating disclosures to policy compliant disclosures. As can be seen, the average time per disclosure *decreases* slightly when more disclosures are violations. This is because the main body of φ_{HIPAA} contains conjunctions after the outermost quantifier. For an instance of the quantifier corresponding to a policy violation, reduce terminates as soon as any one formula in the conjunction reduces to false. On the other hand, for an instance of the quantifier corresponding to a policy compliant disclosure, the entire formula must be reduced.

Consequently, an experiment without policy violations (e.g., Experiment 7) measures the worst case performance of audit. Even this performance is very practical — for a database of size 15MB containing 21,684 disclosures without any violations, reduce can audit each disclosure in an average time of 0.12s.

In the second set of experiments, we simulate a run of our algorithm with three iterations on progressively larger logs, and measure the size of the residual formula after each iteration. The input to the first iteration is the formula φ_{HIPAA} and a log containing 5401 disclosures.

Figure 3 summarizes the evaluation results. The first column is the size of a textual rendering of the policy formula that is the input of reduce, and the second column is the size of the resulting output formula. The remaining columns document the running times, memory consumption, the size of the log in an iteration, and the number of disclosures in the log. The output formulas are quite large, e.g., after the first run, the output formula has a size 4,654.07 KB. The output formula is large because for each disclosure in the log, most of the policy formula is replicated in the output, but owing to a large number of SUBJ atoms in φ_{HIPAA}, these replicated formulas are not eliminated completely (67 out of the 84 clauses contain SUBJ atoms [15]). Although large output formulas slow subsequent iterations, such automatic iterations are not the intended application of reduce. Instead, we expect that our reduce algorithm will be used with an interactive front-end that allows an auditor to select specific subformulas of interest (e.g., those pertaining to disclosures by a specific agent) and run the reduce algorithm only on them. The front-end may also allow the auditor to provide information about subjective atoms and record such information for subsequent use.

6. RELATED WORK

Runtime Monitoring with Temporal Logic. A lot of prior work addresses the problem of *runtime monitoring* of policies expressed in Linear Temporal Logic (LTL) [32, 3, 29, 31, 9, 5] and its extensions [30, 5, 31]. Although similar in the spirit of enforcing policies, the intended deployment of our work is different: We assume that system logs are accumulated independently and given to our algorithm, whereas an integral component of runtime monitoring is accumulation of system logs on the fly. Our assumption about the availability of system logs fits practical situations like health organizations, which collect transmission, disclosure and other logs to comply with regulations such as HIPAA even if no computerized policy enforcement mechanism is in place.

Comparing only the expressiveness of the logic, our work is more advanced than all existing work on policy enforcement. First, LTL can be encoded in our logic easily [15]. Second, we allow expressive quantification in our logic, whereas prior work is either limited to propositional logic [32, 3, 29], or, when quantifiers are considered, they are severely restricted [30, 5, 31]. A recent exception to such syntactic restrictions is the work of Basin et al. [9], to which we compare in detail below. Third, no prior work considers the possibility of incompleteness in structures, which our reduce algorithm takes into account.

Recent work by Basin et al. [9] considers runtime monitoring over an expressive fragment of Metric First-order Temporal Logic. Similar to our work, Basin et al. allow quantification over infinite domains, and use a form of mode analysis (called a safe-range analysis) to ensure finiteness during en-

Policy size (KB)	Residual policy size (KB)	Ave. Time per disclosure(s)	Total time (s)	Memory used (KB)	Log size (MB)	Number of disclosures
39.52	4654.07	0.04	209.68	662080	3.98	5401
4654.07	9369.32	0.75	8193.76	4063472	7.68	10866
9369.32	19186.84	2.99	65009.30	4033484	15.1	21742

Figure 3: Experimental evaluation of iterative runs of the algorithm reduce

forcement. However, Basin et al.'s mode analysis is weaker than ours; in particular, it cannot relate the same variable in the input and output positions of two different conjuncts of a restriction and requires that each free variable appear in at least one predicate with a finite model. As a consequence, some policies such as the HIPAA policy φ_{HIPAA} from Section 5, whose top-level restriction $(\texttt{send}(p_1, p_2, m, \tau) \wedge \texttt{purp}(m, u))$ contains a variable u not occurring in any predicate with a finite model, cannot be enforced in their framework, but can be enforced in ours.

Formal Frameworks for Policy Audit. Cederquist et al. [12] present a proof-based system for a-posteriori audit, where policy obligations are discharged by constructing formal proofs. The leaves of proofs are established from logs, but the audit process only checks that an obligation has been satisfied somewhere in the past. Further, there is no systematic mechanism to instantiate quantifiers in proofs. However, using connectives of linear logic, the mechanism admits policies that rely on use-once permissions.

Iterative Enforcement. The idea of iteratively rewriting the policy over evolving logs has been considered previously [29, 32], but only for propositional logic where the absence of quantifiers simplifies the problem considerably. Bauer et al. [3] use a different approach for iterative enforcement: they convert an LTL formula with limited first-order quantification to a Büchi automaton and check whether the automaton accepts the input log. Further, they also use a three-valued semantic model similar to ours, but assume that logs record all information about past events (past-completeness). Three-valued structures have also been considered in work on generalized model checking [11, 19]. However, the problems addressed in that line of work are different; the objective there is to check whether there exist extensions of a given structure in which a formula is satisfied (or falsified).

Compliance Checking. Barth et al. [6] present two formal definitions of compliance of an action with a policy, called strong and weak compliance. An action is *strongly compliant* with a policy given a trace if there exists an extension of the trace that contains the action and satisfies the policy. We do not consider strong compliance in this paper. An action is *weakly compliant* with a policy in Propositional LTL (PLTL) given a trace if the trace augmented with the action satisfies the present requirements of the policy. However, a weakly compliant action might incur unsatisfiable future requirements. The technical definition is stated in terms of a standard tableau construction for PLTL [24] that syntactically separates present and future requirements. Our correctness property for reduce generalizes weak compliance to a richer class of policies and structures: PLTL can be encoded in our policy logic, the residual formula generalizes future requirements, and past-complete traces are a special case of our partial structures.

In a related paper, Barth et al. [7] present an algorithm that examines audit logs to detect policy violations and identify agents to blame for policy violations. While our audit algorithm can be used to detect violations of a much richer class of policies than the propositional logics considered by Barth et al., it does not identify agents to be blamed for violations.

Lam et al. [22] represent policy requirements of a part of the HIPAA Privacy Rule in an extension of Prolog with stratified negation, called pLogic, and use it to implement a compliance checker for a medical messaging system. The compliance checker makes decisions about legitimacy of messages entering the system based on eight attributes attached to each message (such as its sender, intended recipient, subject, type of information and purpose). The prototype tool has a usable front-end and provides a useful interface for understanding what types of disclosures and uses of personal health information are permitted and forbidden by the HIPAA Privacy Rule. However, as recognized by the authors, the approach has certain limitations in demonstrating compliance with the HIPAA Privacy Rule. First, it does not support temporal conditions. While pLogic uses specialized predicates to capture that certain events happened in the past, it cannot represent future obligations needed to formalize many clauses in HIPAA. In contrast, our policy logic and the reduce algorithm handle temporal conditions, including real-time conditions. Second, reasoning in pLogic proceeds assuming that all asserted beliefs, purposes and types of information associated with messages are correct. In contrast, since reduce mines logs to determine truth values of atoms, it does not assume facts unless there is evidence in logs to back them up. Typically, a purpose or belief will be taken as true only if a human auditor (or some other oracle) supplies evidence to that effect. Finally, our prototype implementation was evaluated with a formalization of the entire HIPAA Privacy Rule, whereas Lam et al. formalize only §§164.502, 164.506 and 164.510.

Policy Specification and Analysis. Several variants of LTL have been used to *specify* the properties of programs, business processes and security and privacy policies [6, 15, 8, 18, 23]. The logic we use as well as the formalization of HIPAA used in our experiments are adapted from our prior work on the logic PrivacyLFP [15]. PrivacyLFP, in turn, draws inspiration from earlier work on the logic LPU [6]. However, PrivacyLFP is more expressive than LPU because it allows first-order quantification over infinite domains.

Further, several access-control models have extensions for specifying usage control and future obligations [20, 10, 28, 21, 26, 16, 27]. Some of these models assume a pre-defined notion of obligations [21, 26]. For instance, Irwin et al. [21] model obligations as tuples containing the subject of the obligation, the actions to be performed, the objects that are targets of the actions and the time frames of the obligations. Other models leave specifications for obligations

abstract [20, 10, 28]. Such specific models and the ensuing policies can be encoded in our logic using quantifiers.

There also has been much work on analyzing the properties of policies represented in formal models. For instance, Ni et al. study the interaction between obligation and authorization [26], Irwin et al. have analyzed accountability problems with obligations [21], and Dougherty et al. have modeled the interaction between obligations and programs [16]. These methods are orthogonal to our objective of policy enforcement.

Finally, privacy languages such as EPAL [4] and privacyAPI [25] do not include obligations or temporal modalities as primitives, and are less expressive than our framework.

7. CONCLUSION AND FUTURE WORK

We have presented the design, implementation, and evaluation of a provably correct iterative algorithm for policy audit, reduce, that works even with incomplete audit logs. Our policy logic is expressive enough to represent real privacy legislation like HIPAA, yet tractable due to a carefully designed static analysis. Our empirical evaluation shows that reduce is efficient enough to be used in practice.

In future work, we plan to investigate two applications besides after-the-fact auditing using the reduce algorithm as a core. The first application is *runtime monitoring* of policies. In this context, reduce can be applied to the part of the policy relevant to an action to be performed with a hypothetical log that includes the future action. If the resulting formula is unsatisfiable, then the action to be performed is a violation. If the resulting formula is satisfied, then the action is permitted. Finally, if reduce outputs a non-trivial residual formula (involving, for example, beliefs, purposes, or future obligations), the residual policy can be used to guide agents about legitimacy of actions they are about to perform. Such a tool will be useful to organizations in educating their employees about appropriate disclosures and uses of personal information as described in complex policies, such as the HIPAA Privacy Rule.

The second application is *accounting* of actions involving personal information of individual data subjects. Proposals for informing patients about disclosures and uses of their personal health information are currently being debated in the U.S. [14]. reduce can be run on the entire policy with a subset of the logs that pertain to a specific agent to discover all disclosures related to that agent and evidence supporting whether the disclosures were violations, permitted, or conditionally permitted.

We also plan to integrate our audit algorithm into a policy-aware health information exchange system that is being developed as part of the SHARPS project (`http://sharps.org`) that we participate in. Ensuring that disclosures of protected health information are made in accordance with privacy regulations is critical in this setting. This project also provides a vehicle to deploy and evaluate the effectiveness of this algorithm over real hospital logs. Another direction of ongoing and future work is to develop semantic foundations and enforcement techniques for concepts in privacy policies related to purposes and beliefs that at first glance appear difficult to formalize and enforce using computational methods.

Acknowledgments. This work was partially supported by the U.S. Army Research Office contract "Perpetually Available and Secure Information Systems" (DAAD19-02-1-0389) to Carnegie Mellon CyLab, the NSF Science and Technology Center TRUST, the NSF CyberTrust grant "Privacy, Compliance and Information Risk in Complex Organizational Processes", the AFOSR MURI "Collaborative Policies and Assured Information Sharing", and HHS Grant no. HHS 90TR0003/01.

8. REFERENCES

[1] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.

[2] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[3] F. Baader, A. Bauer, and M. Lippmann. Runtime verification using a temporal description logic. In *Proceedings of the 7th International Conference on Frontiers of Combining Systems (FroCos)*, pages 149–164, 2009.

[4] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, LNCS 2808, pages 101–119, 2003.

[5] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 44–57, 2004.

[6] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)*, pages 184–198, 2006.

[7] A. Barth, J. C. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 279–294, 2007.

[8] D. Basin, F. Klaedtke, and S. Müller. Monitoring security policies with metric first-order temporal logic. In *Proceeding of the 15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 23–34, 2010.

[9] D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 1–18, 2010.

[10] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *Journal of Network and Systems Management*, 11:351–372, 2003.

[11] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, pages 168–182, 2000.

[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*, 6(2):133–151, 2007.

[13] Deloitte & Touche and the Ponemon Institute.

Enterprise@Risk: 2007 Privacy and Data Protection Survey. White Paper, December 2007.

[14] Department of Health and Human Services, Office of the Secretary. HIPAA Privacy Rule accounting of disclosures under the health information technology for economic and clinical health act. 45 CFR 164, 2011. Available at `http://www.gpo.gov/fdsys/pkg/FR-2011-05-31/pdf/2011-13297.pdf`.

[15] 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 (WPES)*, 2010. Full version: Carnegie Mellon University Technical Report CMU-CyLab-10-007.

[16] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS)*, pages 375–389, 2007.

[17] D. Garg, L. Jia, and A. Datta. A logical method for policy enforcement over evolving audit logs. Technical Report CMU-CyLab-11-002, Carnegie Mellon University, 2011.

[18] C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou. Regulations expressed as logical models (REALM). In *Proceeding of the 18th Annual Conference on Legal Knowledge and Information Systems (JURIX)*, pages 37–48, 2005.

[19] P. Godefroid and M. Huth. Model checking vs. generalized model checking: Semantic minimizations for temporal logics. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 158–167, 2005.

[20] M. Hilty, D. A. Basin, and A. Pretschner. On obligations. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, pages 98–117, 2005.

[21] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 134–143, 2006.

[22] P. E. Lam, J. C. Mitchell, and S. Sundaram. A formalization of HIPAA for a medical messaging system. In *Proceedings of the 6th International Conference on Trust, Privacy and Security in Digital Business (TrustBus)*, pages 73–85, 2009.

[23] Y. Liu, S. Müller, and K. Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46:335–361, 2007.

[24] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[25] M. J. May, C. A. Gunter, and I. Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations (CSFW)*, pages 85–97, 2006.

[26] Q. Ni, E. Bertino, and J. Lobo. An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 133–142, 2008.

[27] OASIS XACML Committee. Extensible access control markup language (XACML) v2.0, 2004. Available at `http://www.oasis-open.org/specs/#xacmlv2.0`.

[28] J. Park and R. Sandhu. Towards usage control models: beyond traditional access control. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–64, 2002.

[29] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12:151–197, 2005.

[30] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSF)*, pages 220–236, 2001.

[31] O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144:91–108, 2006.

[32] P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[33] US Congress. Gramm-Leach-Bliley Act, Financial Privacy Rule. 15 USC §6801–§6809, November 1999. Available at `http://www.law.cornell.edu/uscode/usc_sup_01_15_10_94_20_I.html`.

[34] US Congress. Health Insurance Portability and Accountability Act of 1996, Privacy Rule. 45 CFR 164, 2002. Available at `http://www.access.gpo.gov/nara/cfr/waisidx_07/45cfr164_07.html`.