Correct Audit Logging: Theory and Practice

Sepehr Amir-Mohammadian[1], Stephen Chong[2], and Christian Skalka[1]

[1] University of Vermont, {samirmoh,ceskalka}@uvm.edu
[2] Harvard University, chong@seas.harvard.edu

Abstract. Retrospective security has become increasingly important to the theory and practice of cyber security, with auditing a crucial component of it. However, in systems where auditing is used, programs are typically instrumented to generate audit logs using manual, ad-hoc strategies. This is a potential source of error even if log analysis techniques are formal, since the relation of the log itself to program execution is unclear. This paper focuses on provably correct program rewriting algorithms for instrumenting formal logging specifications. Correctness guarantees that the execution of an instrumented program produces sound and complete audit logs, properties defined by an information containment relation between logs and the program's logging semantics. We also propose a program rewriting approach to instrumentation for audit log generation, in a manner that guarantees correct log generation even for untrusted programs. As a case study, we develop such a tool for OpenMRS, a popular medical records management system, and consider instrumentation of break the glass policies.

1 Introduction

Retrospective security is the enforcement of security, or detection of security violations, after program execution [33, 36, 40]. Many real-world systems use retrospective security. For example, the financial industry corrects errors and fraudulent transactions not by proactively preventing suspicious transactions, but by retrospectively correcting or undoing these problematic translations. Another example is a hospital whose employees are trusted to access confidential patient records, but who might (rarely) violate this trust [17]. Upon detection of such violations, security is enforced retrospectively by holding responsible employees accountable [41].

Retrospective security cannot be achieved entirely by traditional computer security mechanisms, such as access control, or information-flow control. Reasons include that detection of violations may be external to the computer system (such as consumer reports of fraudulent transactions, or confidential patient information appearing in news media), the high cost of access denial (e.g., preventing emergency-room physicians from accessing medical records) coupled with high trust of systems users (e.g., users are trusted employees that rarely violate this trust) [42]. In addition, remediation actions to address violations may also be external to the computer system, such as reprimanding employees, prosecuting law suits, or otherwise holding users accountable for their actions [41].

Auditing underlies retrospective security frameworks and has become increasingly important to the theory and practice of cyber security. By maintaining a record of appropriate aspects of a computer system's execution, an audit log (and subsequent examination of the audit log) can enable detection of violations, provide sufficient evidence

to hold users accountable for their actions, and support other remediation actions. For example, an audit log can be used to determine *post facto* which users performed dangerous operations, and can provide evidence for use in litigation.

However, despite the importance of auditing to real-world security, relatively little work has focused on the formal foundations of auditing, particularly with respect to defining and ensuring the correctness of audit log generation. Indeed, correct and efficient audit log generation poses at least two significant challenges. First, it is necessary to record sufficient and correct information in the audit log. If a program is manually instrumented, it is possible for developers to fail to record relevant events. Recent work showed that major health informatics systems do not log sufficient information to determine compliance with HIPAA policies [30]. Second, an audit log should ideally not contain more information than needed. While it is straightforward to collect sufficient information by recording essentially *all* events in a computer system, this can cause performance issues, both slowing down the system due to generating massive audit logs, and requiring the handling of extremely large audit logs. Excessive data collection is a key challenge for auditing [23, 14, 29], and is a critical factor in the design of tools that generate and employ audit logs (e.g., spam filters [15]).

A main goal of this paper is to establish formal conditions for audit logs, that can be used to establish correctness conditions for logging instrumentation. We define a general semantics of audit logs using the theory of *information algebra* [32], and interpret both program execution traces and audit logs as information elements. A *logging specification* defines the intended relation between the information in traces and in audit logs. An audit log is correct if it satisfies this relation. A benefit of this formulation is that it separates logging specifications from programs, rather than burying them in code and implementation details.

Separating logging specifications from programs allows a clean declaration of what instrumentation should accomplish, and enables algorithms for implementing general classes of logging specifications that are provably correct. As we will show, correct instrumentation of logging specifications is a safety property, hence enforceable by security automata [38]. Inspired by related approaches to security automata implementation [21], we focus on program rewriting to automatically enforce correct audit instrumentation. Program rewriting has a number of practical benefits versus, for example, program monitors, such as lower OS process management overhead.

We consider a case study of our approach, a program rewriting algorithm for correct instrumentation of logging specifications in OpenMRS (`openmrs.org`), a popular open source medical records software system. Our tool allows system administrators to define logging specifications which are automatically instrumented in OpenMRS legacy code. Implementation details and optimizations are handled transparently by the general program rewriting algorithm, not the logging specification. Formal foundations ensure that logging specifications are implemented correctly by the algorithm. In particular, we show how our system can implement "break the glass" auditing policies.

1.1 A Motivating Example from Practice

Although audit logs contain information *about* program execution, they are not just a straightforward selection of program events. Illustrative examples from practice in-

clude so-called "break the glass policies" used in electronic medical record systems [35]. These policies use access control to disallow care providers from performing sensitive operations such as viewing patient records, however care providers can "break the glass" in an emergency situation to temporarily raise their authority and access patient records, *with the understanding that subsequent sensitive operations will be logged and potentially audited*. One potential accountability goal is the following:

> *In the event that a patient's sensitive information is inappropriately leaked, determine who accessed a given patient's files due to "breaking the glass."*

Since it cannot be predicted a priori whose information may leak, this goal can be supported by using an audit log that records all reads of sensitive files following glass breaking. To generate correct audit logs, programs must be instrumented for logging appropriately, i.e., to implement the following *logging specification* that we call LS_H:

> LS_H : *Record in the log all patient information file reads following a break the glass event, along with the identity of the user that broke the glass.*

If at some point in time in the future it is determined that a specific patient **P**'s information was leaked, logs thus generated can be analyzed with the following query that we call LQ_H:

> LQ_H : *Retrieve the identity of all users that read **P**'s information files.*

The specification LS_H and the query LQ_H together constitute an auditing policy that directly supports the above-stated accountability goal. Their separation is useful since at the time of execution the information leak is unknown, hence **P** is not known. Thus while it is possible to implement LS_H as part of program execution, LQ_H must be implemented retrospectively.

It is crucial to the enforcement of the above accountability goal that LS_H is implemented correctly. If logging is incomplete then some potential recipients may be missed. If logging is overzealous then bloat is possible and audit logs become "write only". These types of errors are common in practice [30]. To establish formal correctness of instrumentation for audit logs, it is necessary to define a formal language of logging specifications, and establish techniques to guarantee that instrumented programs satisfy logging specifications. That is the focus of this paper. Other work has focused on formalisms for querying logs [39, 18], however these works presuppose correctness of audit logs for true accountability.

1.2 Threat Model

With respect to program rewriting (i.e., automatic techniques to instrument existing programs to satisfy a logging specification), we regard the program undergoing instrumentation as untrusted. That is, the program source code may have been written to avoid, confuse, or subvert the automatic instrumentation techniques. We do, however, assume that the source code is well-formed (valid syntax, well-typed, etc.). Moreover, we trust the compiler, the program rewriting algorithm, and the runtime environment in which the instrumented program will ultimately be executed. Confidentiality and non-malleability of generated audit logs, while important, is beyond the scope of this paper.

2 A Semantics of Audit Logging

Our goal in this Section is to formally characterize logging specifications and correctness conditions for audit logs. To obtain a general model, we leverage ideas from the theory of *information algebra* [32], which is an abstract mathematical framework for information systems. In short, we interpret program traces as information, and logging specifications as functions from traces to information. This separates logging specifications from their implementation in code, and defines exactly the information that should be in an audit log. This in turn establishes correctness conditions for audit logging implementations.

Following [38], an *execution trace* $\tau = \kappa_0\kappa_1\kappa_2\ldots$ is a possibly infinite sequence of configurations κ that describe the state of an executing program. We deliberately leave configurations abstract, but examples abound and we explore a specific instantiation for a λ-calculus in Section 4. Note that an execution trace τ may represent the partial execution of a program, i.e. the trace τ may be extended with additional configurations as the program continues execution. We use metavariables τ and σ to range over traces.

An *information algebra* contains information elements X (e.g. a set of logical assertions) taken from a set Φ (the algebra). A partial ordering is induced on Φ by the so-called *information ordering* relation $\leq$, where intuitively for $X, Y \in \Phi$ we have $X \leq Y$ iff Y contains at least as much information as X, though its precise meaning depends on the particular algebra. We say that X and Y are *information equivalent*, and write $X = Y$, iff $X \leq Y$ and $Y \leq X$. We assume given a function $\lfloor\cdot\rfloor$ that is an injective mapping from traces to Φ. This mapping *interprets a given trace as information*, where the injective requirement ensures that information is not lost in the interpretation. For example, if σ is a proper prefix of τ and thus contains strictly less information, then formally $\lfloor\sigma\rfloor \leq \lfloor\tau\rfloor$. We intentionally leave both Φ and $\lfloor\cdot\rfloor$ underspecified for generality, though application of our formalism to a particular logging implementation requires instantiation of them. We discuss an example in Section 3.

We let LS range over *logging specifications*, which are functions from traces to Φ. As for Φ and $\lfloor\cdot\rfloor$, we intentionally leave the language of specifications abstract, but consider a particular instantiation in Section 3. Intuitively, $LS(\tau)$ denotes the information that should be recorded in an audit log during the execution of τ given specification LS, regardless of whether τ actually records any log information, correctly or incorrectly. We call this the semantics of the logging specification LS.

We assume that auditing is implementable, requiring at least that all conditions for logging any piece of information must be met in a finite amount of time. As we will show, this restriction implies that correct logging instrumentation is a safety property [38].

Definition 1. *We require of any logging specification LS that for all traces τ and information $X \leq LS(\tau)$, there exists a finite prefix σ of τ such that $X \leq LS(\sigma)$.*

It is crucial to observe that some logging specifications may *add* information not contained in traces to the auditing process. Security information not relevant to program execution (such as ACLs), interpretation of event data (statistical or otherwise), etc., may be added by the logging specification. For example, in the OpenMRS system, logging of sensitive operations includes a human-understandable "type" designation

which is not used by any other code. Thus, given a trace τ and logging specification LS, it is *not* necessarily the case that $LS(\tau) \leq \lfloor \tau \rfloor$. Audit logging is not just a filtering of program events.

2.1 Correctness Conditions for Audit Logs

A logging specification defines what information should be contained in an audit log. In this section we develop formal notions of *soundness* and *completeness* as audit log correctness conditions. We use metavariable $\mathbb{L}$ to range over audit logs. Again, we intentionally leave the language of audit logs unspecified, but assume that the function $\lfloor \cdot \rfloor$ is extended to audit logs, i.e. $\lfloor \cdot \rfloor$ is an injective mapping from audit logs to Φ. Intuitively, $\lfloor \mathbb{L} \rfloor$ denotes the information in $\mathbb{L}$, interpreted as an element of Φ.

An audit log $\mathbb{L}$ is sound with respect to a logging specification LS and trace τ if the log information is contained in $LS(\tau)$. Similarly, an audit log is complete with respect to a logging specification if it contains all of the information in the logging specification's semantics. Crucially, both definitions are independent of the implementation details that generate $\mathbb{L}$.

Definition 2. *Audit log $\mathbb{L}$ is* sound with respect to logging specification LS and execution trace τ *iff* $\lfloor \mathbb{L} \rfloor \leq LS(\tau)$. *Similarly, audit log $\mathbb{L}$ is* complete with respect to logging specification LS and execution trace τ *iff* $LS(\tau) \leq \lfloor \mathbb{L} \rfloor$.

The relation to log queries. As discussed in Section 1.1, we make a distinction between logging specifications such as LS_H which define how to record logs, and log queries such as LQ_H which ask questions of logs, and our notions of soundness and completeness apply strictly to logging specifications. However, any logging query must assume a logging specification semantics, hence a log that is demonstrably sound and complete provides the same answers on a given query that an "ideal" log would. This is an important property that is discussed in previous work, e.g. as "sufficiency" in [6].

2.2 Correct Logging Instrumentation is a Safety Property

In case program executions generate audit logs, we write $\tau \rightsquigarrow \mathbb{L}$ to mean that a finite trace τ generates $\mathbb{L}$, i.e. $\tau = \kappa_0 \ldots \kappa_n$ and $\mathit{logof}(\kappa_n) = \mathbb{L}$ where $\mathit{logof}(\kappa)$ denotes the audit log in configuration κ, i.e. the residual log after execution of the full trace. Ideally, information that *should* be added to an audit log, *is* added to an audit log, immediately as it becomes available. This ideal is formalized as follows.

Definition 3. *For all logging specifications LS, the trace τ is* ideally instrumented for *LS iff for all finite prefixes σ of τ we have $\sigma \rightsquigarrow \mathbb{L}$ where $\mathbb{L}$ is sound and complete with respect to LS and σ.*

We observe that the restriction imposed on logging specifications by Definition 1, implies that ideal instrumentation of any logging specification is a safety property in the sense defined by Schneider [38][1].

[1] The proofs of Theorems 1-5 in this text are omitted for brevity, but are available in a related Technical Report [3].

Theorem 1. *For all logging specifications LS, the set of ideally instrumented traces is a safety property.*

This result implies that e.g. edit automata can be used to enforce instrumentation of logging specifications (see our Technical Report [3]). However, theory related to safety properties and their enforcement by execution monitors [38, 4] do not provide an adequate semantic foundation for audit log generation, nor an account of soundness and completeness of audit logs.

2.3 Implementing Logging Specifications with Program Rewriting

The above-defined correctness conditions for audit logs provide a foundation on which to establish correctness of logging implementations. Here we consider program rewriting approaches. Since rewriting concerns specific languages, we introduce an abstract notion of programs $\mathfrak{p}$ with an operational semantics that can produce a trace τ. We write $\mathfrak{p} \Downarrow \sigma$ iff program $\mathfrak{p}$ can produce execution trace τ, either deterministically or non-deterministically, and σ is a *finite* prefix of τ.

A rewriting algorithm $\mathcal{R}$ is a (partial) function that takes a program $\mathfrak{p}$ in a source language and a logging specification LS and produces a new program, $\mathcal{R}(\mathfrak{p}, LS)$, in a target language.[2] The intent is that the target program is the result of instrumenting $\mathfrak{p}$ to produce an audit log appropriate for the logging specification LS. A rewriting algorithm may be partial, in particular because it may only be intended to work for a specific set of logging specifications.

Ideally, a rewriting algorithm should preserve the semantics of the program it instruments. That is, $\mathcal{R}$ is semantics-preserving if the rewritten program simulates the semantics of the source code, modulo logging steps. We assume given a correspondence relation $:\approx$ on execution traces. A coherent definition of correspondence should be similar to a bisimulation, but it is not necessarily symmetric nor a bisimulation, since the instrumented target program may be in a different language than the source program. We deliberately leave the correspondence relation underspecified, as its definition will depend on the instantiation of the model. We provide an explicit definition of correspondence for λ-calculus source and target languages in Section 4.

Definition 4. *Rewriting algorithm $\mathcal{R}$ is* semantics preserving *iff for all programs $\mathfrak{p}$ and logging specifications LS such that $\mathcal{R}(\mathfrak{p}, LS)$ is defined, all of the following hold:*

1. *For all traces τ such that $\mathfrak{p} \Downarrow \tau$ there exists τ' with $\tau :\approx \tau'$ and $\mathcal{R}(\mathfrak{p}, LS) \Downarrow \tau'$.*
2. *For all traces τ such that $\mathcal{R}(\mathfrak{p}, LS) \Downarrow \tau$ there exists a trace τ' such that $\tau' :\approx \tau$ and $\mathfrak{p} \Downarrow \tau'$.*

In addition to preserving program semantics, a correctly rewritten program constructs a log in accordance with the given logging specification. More precisely, if LS is a given logging specification and a trace τ describes execution of a source program, rewriting should produce a program with a trace τ' that corresponds to τ (i.e., $\tau :\approx \tau'$),

[2] We use metavariable $\mathfrak{p}$ to range over programs in either the source or target language; it will be clear from context which language is used.

where the log $\mathbb{L}$ generated by τ' contains the same information as $LS(\tau)$, or at least a sound approximation. Some definitions of $:\approx$ may allow several target-language traces to correspond to source-language traces (as for example in Section 4, Definition 10). In any case, we expect that at least one simulation exists. Hence we write $simlogs(\mathfrak{p}, \tau)$ to denote a nonempty set of logs $\mathbb{L}$ such that, given a finite source language trace τ and target program $\mathfrak{p}$, there exists some trace τ' where $\mathfrak{p} \Downarrow \tau'$ and $\tau :\approx \tau'$ and $\tau' \rightsquigarrow \mathbb{L}$. The name $simlogs$ evokes the relation to logs resulting from simulating executions in the target language.

The following definitions then establish correctness conditions for rewriting algorithms. Note that satisfaction of either of these conditions only implies condition (1) of Definition 4, not condition (2), so semantics preservation is an independent condition.

Definition 5. *Rewriting algorithm $\mathcal{R}$ is* sound/complete *iff for all programs $\mathfrak{p}$, logging specifications LS, and finite traces τ where $\mathfrak{p} \Downarrow \tau$, for all $\mathbb{L} \in simlogs(\mathcal{R}(\mathfrak{p}, LS), \tau)$ it is the case that $\mathbb{L}$ is sound/complete with respect to LS and τ.*

3 Languages for Logging Specifications

Now we go into more detail about information algebra and why it is a good foundation for logging specifications and semantics. We use the formalism of information algebras to characterize and compare the information contained in an audit log with the information contained in an actual execution. For a detailed account of information algebra, the reader is referred to a definitive survey paper [32]– available space disallows a detailed account here. In short, in addition to a definition of the elements of Φ, any information algebra Φ includes two basic operators:

- Combination: The operation $X \otimes Y$ *combines* the information in elements $X, Y \in \Phi$.
- Focusing: The operation $X^{\Rightarrow S}$ isolates the elements of $X \in \Phi$ that are relevant to a *sublanguage* S, i.e. the subpart of X specified by S.

Focusing and combination must additionally satisfy certain properties (see our Technical Report [3]). The definitions of elements $X \in \Phi$, sublanguages S, combination, and focusing constitute the definition of the algebra. In all cases, the relation $X \leq Y$ holds iff $X \otimes Y = Y$. Proving that $\otimes$ has been correctly defined for an algebra implies that $\leq$ is a partial order [32].

3.1 Support for Various Approaches

Various approaches are taken to audit log generation and representation, including logical [18], database [1], and probabilistic approaches [43]. Information algebra is sufficiently general to contain relevant systems as instances, so our notions of soundness and completeness can apply broadly. Here we discuss logical and database approaches.

First Order Logic (FOL) Logics have been used in several well-developed auditing systems [24, 10], for the encoding of both audit logs and queries. FOL in particular is attractive due to readily available implementation support, e.g. Datalog and Prolog.

Let Greek letters ϕ and ψ range over FOL formulas and let capital letters X, Y, Z range over sets of formulas. We posit a sound and complete proof theory supporting judgements of the form $X \vdash \phi$. In this text we assume without loss of generality a natural deduction proof theory.

Elements of our algebra are sets of formulas closed under logical entailment. Intuitively, given a set of formulas X, the closure of X is the set of formulas that are logically entailed by X, and thus represents all the information contained in X. In spirit, we follow the treatment of sentential logic as an information algebra explored in related foundational work [32], however our definition of closure is syntactic, not semantic.

Definition 6. *We define a closure operation C, and a set Φ_{FOL} of closed sets of formulas:*

$$C(X) = \{\phi \mid X \vdash \phi\} \qquad \Phi_{FOL} = \{X \mid C(X) = X\}$$

Note in particular that $C(\varnothing)$ is the set of logical tautologies.

Let $Preds$ be the set of all predicate symbols, and let $S \subseteq Preds$ be a set of predicate symbols. We define *sublanguage* L_S to be the set of well-formed formulas over predicate symbols in S (and including boolean atoms T and F, and closed under the usual first-order connectives and binders). We will use sublanguages to define refinement operations in our information algebra. Subset containment induces a lattice structure, denoted $\mathcal{S}$, on the set of all sublanguages, with $\mathcal{F} = L_{Preds}$ as the top element.

Now we can define the focus and combination operators, which are the fundamental operators of an information algebra. Focusing isolates the component of a closed set of formulas that is in a given sublanguage. Combination closes the union of closed sets of formulas. Intuitively, the focus of a closed set of formulas X to sublanguage L is the refinement of the information in X to the formulas in L. The combination of closed sets of formulas X and Y combines the information of each set.

Definition 7. *Define:*

1. *Focusing:* $X^{\Rightarrow S} = C(X \cap L_S)$ *where* $X \in \Phi_{FOL}$, $S \subseteq Preds$
2. *Combination:* $X \otimes Y = C(X \cup Y)$ *where* $X, Y \in \Phi_{FOL}$

These definitions of focusing and combination enjoy a number of properties within the algebra, as stated in the following Theorem, establishing that the construction is a domain-free information algebra [31]. FOL has been treated as an information algebra before, but our definitions of combination and focusing and hence the result are novel.

Theorem 2. *Structure $(\Phi_{FOL}, \mathcal{S})$ with focus operation $X^{\Rightarrow S}$ and combination operation $X \otimes Y$ forms a* domain-free information algebra*.*

In addition, to interpret traces and logs as elements of this algebra, i.e. to define the function $\lfloor \cdot \rfloor$, we assume existence of a function $toFOL(\cdot)$ that injectively maps traces and logs to sets of FOL formulas, and then take $\lfloor \cdot \rfloor = C(toFOL(\cdot))$. To define

the range of $toFOL(\cdot)$, that is, to specify how trace information will be represented in FOL, we assume the existence of *configuration description predicates* P which are each at least unary. Each configuration description predicate fully describes some element of a configuration κ, and the first argument is always a natural number t, indicating the time at which the configuration occurred. A set of configuration description predicates with the same timestamp describes a configuration, and traces are described by the union of sets describing each configuration in the trace. In particular, the configuration description predicates include predicate $\mathrm{Call}(t, f, x)$, which indicates that function f is called at time t with argument x. We will fully define $toFOL(\cdot)$ when we discuss particular source and target languages for program rewriting.

Example 1. We return to the example described in Section 1.1 to show how FOL can express break the glass logging specifications. Adapting a logic programming style, the trace of a program can be viewed as a fact base, and the logging specification LS_H performs resolution of a $\mathrm{LoggedCall}$ predicate, defined via the following Horn clause we call ψ_H:

$$\forall t, d, s, u.(\mathrm{Call}(t, \mathbf{read}, u, d) \wedge \mathrm{Call}(s, \mathbf{breakGlass}, u) \wedge s < t \ \wedge \mathrm{PatientInfo}(d)) \\ \implies \mathrm{LoggedCall}(t, \mathbf{read}, u, d)$$

Here we imagine that $\mathbf{breakGlass}$ is a break the glass function where u identifies the current user and $\mathrm{PatientInfo}$ is a predicate specifying which files contain patient information. The log contains only valid instances of $\mathrm{LoggedCall}$ given a particular trace, which specify the user and sensitive information accessed following glass breaking, which otherwise would be disallowed by a separate access control policy.

Formally, we define logging specifications in a logic programming style by using combination and focusing. Any logging specification is parameterized by a sublanguage S that identifies the predicate(s) to be resolved and Horn clauses X that define it/them, hence we define a functional *spec* from pairs (X, S) to specifications LS, where we use λ as a binder for function definitions in the usual manner:

Definition 8. *The function spec is given a pair* (X, S) *and returns a* FOL logging specification*, i.e. a function from traces to elements of* Φ_{FOL}*:*

$$spec(X, S) = \lambda\tau.(\lfloor\tau\rfloor \otimes C(X))^{\Rightarrow S}.$$

In any logging specification $spec(X, S)$*, we call* X *the* guidelines*.*

The above example LS_H would then be formally defined as $spec(\psi_H, \{\mathrm{LoggedCall}\})$.

Relational Database Relational algebra is a canonical example of an information algebra, though we provide a different formulation than the standard one [32] since the latter is not suited to our purpose here. We define databases D as sets of relations, where a relation X is a set of *tuples*. We write $((a_1 : x_1), ..., (a_n : x_1))$ to denote an n-ary tuple with attributes (aka label) a_i associated with values x_i. Databases are elements of the information algebra, and sublanguages S are collections of sets of attributes. Each set of attributes corresponds to a specific relation. We define focusing as the restriction to particular relations in a database, and combination as the union of databases.

Hence, letting $\leq_{RA}$ denote the relational algebra information ordering, $D_1 \leq_{RA} D_2$ iff $D_1 \otimes D_2 = D_2$. We refer to this algebra as Φ_{RA}. The details of our formulation and the proof that it satisfies the required properties is given in our Technical Report [3]. Relational databases are heavily used for storing and querying audit logs, so this formulation is crucial for practical application of our correctness properties, as discussed in Section 5.

3.2 Transforming and Combining Audit Logs

Multiple audit logs from different sources are often combined in practice. Also, logging information is often transformed for storage and communication. For example, log data may be generated in common event format (CEF), which is parsed and stored in relational database tables, and subsequently exported and communicated via JSON. In all cases, it is necessary to characterize the effect of transformation (if any) on log information, and relate queries on various representations to the logging specification semantics. Otherwise, it is unclear what is the relation of log queries to log-generating programs.

To address this, information algebra provides a useful concept called *monotone mapping*. Given two information algebras Ψ_1 and Ψ_2 with ordering relations $\leq_1$ and $\leq_2$ respectively, a mapping μ from elements X, Y of Ψ_1 to elements $\mu(X), \mu(Y)$ of Ψ_2 is monotone iff $X \leq_1 Y$ implies $\mu(X) \leq_2 \mu(Y)$. For example, assuming that Ψ_1 is our FOL information algebra while Ψ_2 is relational algebra, we can define a monotone mapping using a *least Herbrand interpretation* [11], denoted $\mathfrak{H}$, and by positing a function $attrs$ from n-ary predicate symbols to functions mapping numbers $1, ..., n$ to labels. That is, $attrs(\mathrm{P})(n)$ is the label associated with the nth argument of predicate P. We require that if $\mathrm{P} \neq \mathrm{Q}$ then $attrs(\mathrm{P})(j) \neq attrs(\mathrm{Q})(k)$ for all j, k. To map predicates to tuples we have:

$$tuple(\mathrm{P}(x_1, \ldots, x_n)) = ((attrs(\mathrm{P})(1) : x_1), \ldots, (attrs(\mathrm{P})(n) : x_n))$$

Then to obtain a relation from all valid instances of a particular predicate P given formulas X we define:

$$R_{\mathrm{P}}(X) = \{tuple(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(X)\}$$

Now we define the function rel which is collection of all relations obtained from X, where $\mathrm{P}_1, ..., \mathrm{P}_n$ are the predicate symbols occurring in X:

$$rel(X) = \{R_{\mathrm{P}_1}(X), \cdots, R_{\mathrm{P}_n}(X)\}$$

Theorem 3. *rel is a monotone mapping.*

Thus, if we wish to generate an audit log $\mathbb{L}$ as a set of FOL formulas, but ultimately store the data in a relational database, we are still able to maintain a formal relation between stored logs and the semantics of a given trace τ and specification LS. E.g., if a log $\mathbb{L}$ is sound with respect to τ and LS, then $rel(\lfloor \mathbb{L} \rfloor) \leq_{RA} rel(LS(\tau))$. While the data in $rel(\lfloor \mathbb{L} \rfloor)$ may very well be broken up into multiple relations in practice, e.g. to

compress data and/or for query optimization, the formalism also establishes correctness conditions for the transformation that relate resulting information to the logging semantics $LS(\tau)$ by way of the mapping. We reify this idea in our OpenMRS implementation as discussed in Section 5.2.

4 Rewriting Programs with Logging Specifications

Since correct logging instrumentation is a safety property (2.2), there are several possible implementation strategies. For example, one could define an edit automata that enforces the property (see our Technical Report [3]), that could be implemented either as a separate program monitor or using IRM techniques [21]. But since we are interested in program rewriting for a particular class of logging specifications, the approach we discuss here is more simply stated and proven correct than a general IRM methodology.

We specify a class of logging specifications of interest, along with a program rewriting algorithm that is sound and complete for it. We consider a basic λ-calculus that serves as formal setting to establish correctness of a program rewriting approach to correct instrumentation of logging specification. We use this same approach to implement an auditing tool for OpenMRS, described in the next Section. The supported class of logging specifications is predicated on temporal properties of function calls and characteristics of their arguments. This class has practical potential since security-sensitive operations are often packaged as functions or methods (e.g. in medical records software [37]), and the supported class allows complex policies such as break the glass to be expressed. The language of logging specifications is FOL, and we use Φ_{FOL} to define the semantics of logging and prove correctness of the algorithm.

4.1 Source Language

We first define a source language Λ_{call}, including the definitions of configurations, execution traces, and function $toFOL(\cdot)$ that shows how we concretely model execution traces in FOL.

Language Λ_{call} is a simple call-by-value λ-calculus with named functions. A Λ_{call} program is a pair $(e, \mathcal{C})$ where e is an expression, and $\mathcal{C}$ is a *codebase* which maps function names to function definitions. A Λ_{call} configuration is a triple $(e, n, \mathcal{C})$, where e is the expression remaining to be evaluated, n is a timestamp (a natural number) that indicates how many steps have been taken since program execution began, and $\mathcal{C}$ is a codebase. The codebase does not change during program execution.

The syntax of Λ_{call} is as follows.

$$
\begin{array}{rr}
v ::= x \mid \mathtt{f} \mid \lambda x.\, e & \textit{values} \\
e ::= e\ e \mid v & \textit{expressions} \\
E ::= [\,] \mid E\ e \mid v\ E & \textit{evaluation contexts} \\
\kappa ::= (e, n, \mathcal{C}) & \textit{configurations} \\
\mathfrak{p} ::= (e, \mathcal{C}) & \textit{programs}
\end{array}
$$

The small-step semantics of Λ_{call} is defined as follows.

$$\frac{}{((\lambda x.\ e)\ v, n, \mathcal{C}) \to (e[v/x], n+1, \mathcal{C})}\ \beta \qquad \frac{\mathcal{C}(\mathbf{f}) = \lambda x.\ e}{(\mathbf{f}\ v, n, \mathcal{C}) \to (e[v/x], n+1, \mathcal{C})}\ \beta_{\text{Call}}$$

$$\frac{(e, n, \mathcal{C}) \to (e', n', \mathcal{C})}{(E[e], n, \mathcal{C}) \to (E[e'], n', \mathcal{C})}\ \text{Context}$$

An execution trace τ is a sequence of configurations, and for a program $\mathfrak{p} = (e, \mathcal{C})$ and execution trace $\tau = \kappa_0 \dots \kappa_n$ we define $\mathfrak{p} \Downarrow \tau$ if and only if $\kappa_0 = (e, 0, \mathcal{C})$ and for all $i \in 1..n$ we have $\kappa_{i-1} \to \kappa_i$.

We now show how to model a configuration as a set of ground instances of predicates, and then use this to model execution traces. We posit predicates Call, App, Value, Context, and Codebase to logically denote run time entities. For $\kappa = (e, n, \mathcal{C})$, we define $toFOL(\kappa)$ by cases, where $\langle \mathcal{C} \rangle_n = \bigcup_{\mathbf{f} \in \text{dom}(\mathcal{C})} \{\text{Codebase}(n, \mathbf{f}, \mathcal{C}(\mathbf{f}))\}$[3].

$$\begin{aligned} toFOL(v, n, \mathcal{C}) &= \{\text{Value}(n, v)\} \cup \langle \mathcal{C} \rangle_n \\ toFOL(E[\mathbf{f}\ v], n, \mathcal{C}) &= \{\text{Call}(n, \mathbf{f}, v), \text{Context}(n, E)\} \cup \langle \mathcal{C} \rangle_n \\ toFOL(E[(\lambda x.\ e)\ v)], n, \mathcal{C}) &= \{\text{App}(n, (\lambda x.e), v), \text{Context}(n, E)\} \cup \langle \mathcal{C} \rangle_n \end{aligned}$$

We define $toFOL(\tau)$ for a potentially infinite execution trace $\tau = \kappa_0 \kappa_1 \dots$ by defining it over its prefixes. Let $\text{prefix}(\tau)$ denote the set of prefixes of τ. Then, $toFOL(\tau) = \bigcup_{\sigma \in \text{prefix}(\tau)} toFOL(\sigma)$, where $toFOL(\sigma) = toFOL(\kappa_0) \cup \dots \cup toFOL(\kappa_n)$, for $\sigma = \kappa_0 \dots \kappa_n$. Function $toFOL(\cdot)$ is injective up to α-equivalence since $toFOL(\tau)$ fully and uniquely describes the execution trace τ.

4.2 Specifications Based on Function Call Properties

We define a class **Calls** of logging specifications that capture temporal properties of function calls, such as those reflected in break the glass policies. We restrict specification definitions to safe Horn clauses to ensure applicability of well-known results and total algorithms such as Datalog [11]. Specifications in **Calls** support logging of calls to a specific function $\mathbf{f}$ that happen after functions $\mathbf{g}_1, \dots, \mathbf{g}_n$ are called. Conditions on all function arguments, and times of their invocation, can be defined via a predicate ϕ. Hence more precise requirements can be imposed, e.g. a linear ordering on function calls, particular values of functions arguments, etc.

Definition 9. **Calls** *is the set of all logging specifications* $spec(X, \{\text{LoggedCall}\})$ *where* X *contains a safe Horn clause of the following form:*

$$\forall t_0, \dots, t_n, x_0, \dots, x_n\ .\ \text{Call}(t_0, \mathbf{f}, x_0) \bigwedge_{i=1}^{n} (\text{Call}(t_i, \mathbf{g}_i, x_i) \wedge t_i < t_0) \wedge \phi((x_0, t_0), \dots, (x_n, t_n)) \implies \text{LoggedCall}(t_0, \mathbf{f}, x_0).$$

[3] While Λ_{call} expressions and evaluation contexts appear as predicate arguments, their syntax can be written as string literals to conform to typical Datalog or Prolog syntax.

While set X may contain other safe Horn clauses, in particular definitions of predicates occurring in ϕ, no other Horn clause in X uses the predicate symbols LoggedCall*,* Value*,* Context*,* Call*,* App*, or* Codebase*. For convenience in the following, we define* $Logevent(LS) = \mathbf{f}$ *and* $Triggers(LS) = \{\mathbf{g}_1, ..., \mathbf{g}_n\}$.

We note that specifications in **Calls** clearly satisfy Definition 1, since preconditions for logging a particular call to $\mathbf{f}$ must be satisfied at the time of that call.

4.3 Target Language

The syntax of target language $\Lambda_{\log}$ extends Λ_{call} syntax with a command to track *logging preconditions* ($callEvent(\mathbf{f}, v)$), i.e. calls to logging triggers, and a command to emit log entries ($emit(\mathbf{f}, v)$). Configurations are extended to include a set X of logging preconditions, and an audit log $\mathbb{L}$.

$$
\begin{array}{ll}
e ::= \ldots \mid callEvent(\mathbf{f}, v); e \mid emit(\mathbf{f}, v); e & \textit{expressions} \\
\kappa ::= (e, X, n, \mathbb{L}, \mathcal{C}) & \textit{configurations}
\end{array}
$$

The semantics of $\Lambda_{\log}$ extends the semantics of Λ_{call} with new rules for commands $callEvent(\mathbf{f}, v)$ and $emit(\mathbf{f}, v)$, which update the set of logging preconditions and audit log respectively. An instrumented program uses the set of logging preconditions to determine when it should emit events to the audit log. The semantics is parameterized by a guideline $X_{Guidelines}$, typically taken from a logging specification. Given the definition of **Calls**, these semantics would be easy to implement using e.g. a Datalog proof engine.

Precondition

$$\frac{}{(callEvent(\mathbf{f}, v); e, X, n, \mathbb{L}, \mathcal{C}) \to (e, X \cup \{\text{Call}(n-1, \mathbf{f}, v)\}, n, \mathbb{L}, \mathcal{C})}$$

Log

$$\frac{X \cup X_{Guidelines} \vdash \text{LoggedCall}(n-1, \mathbf{f}, v)}{(emit(\mathbf{f}, v); e, X, n, \mathbb{L}, \mathcal{C}) \to (e, X, n, \mathbb{L} \cup \{\text{LoggedCall}(n-1, \mathbf{f}, v)\}, \mathcal{C})}$$

NoLog

$$\frac{X \cup X_{Guidelines} \nvdash \text{LoggedCall}(n-1, \mathbf{f}, v)}{(emit(\mathbf{f}, v); e, X, n, \mathbb{L}, \mathcal{C}) \to (e, X, n, \mathbb{L}, \mathcal{C})}$$

Note that to ensure that these instrumentation commands do not change execution behavior, the configuration's time is not incremented when $callEvent(\mathbf{f}, v)$ and $emit(\mathbf{f}, v)$ are evaluated. That is, the configuration time counts the number of source language computation steps.

The rules Log and NoLog rely on checking whether $X_{Guidelines}$ and logging preconditions X entail $\text{LoggedCall}(n-1, \mathbf{f}, v)$. For a target language program $\mathfrak{p} = (e, \mathcal{C})$ and execution trace $\tau = \kappa_0 \ldots \kappa_n$ we define $\mathfrak{p} \Downarrow \tau$ if and only if $\kappa_0 = (e, \emptyset, 0, \emptyset, \mathcal{C})$ and for all $i \in 1..n$ we have $\kappa_{i-1} \to \kappa_i$.

To establish correctness of program rewriting, we need to define a correspondence relation $:\approx$. Source language execution traces and target language execution traces correspond if they represent the same expression evaluated to the same point. We make special cases for when the source execution is about to perform a function application that the target execution will track or log via an $callEvent(\mathbf{f}, v)$ or $emit(\mathbf{f}, v)$ command. In these cases, the target execution may be ahead by one or two steps, allowing time for addition of information to the log.

Definition 10. *Given source language execution trace $\tau = \kappa_0 \dots \kappa_m$ and target language execution trace $\tau' = \kappa'_0 \dots \kappa'_n$, where $\kappa_i = (e_i, t_i, \mathcal{C}_i)$ and $\kappa'_i = (e'_i, X_i, t'_i, \mathbb{L}_i, \mathcal{C}'_i)$, $\tau :\approx \tau'$ iff $e_0 = e'_0$ and either*

1. *$e_m = e'_n$ (taking $=$ to mean syntactic equivalence); or*
2. *$e_m = e'_{n-1}$ and $e'_n = callEvent(\mathbf{f}, v); e'$ for some expressions $\mathbf{f}$, v, and e'; or*
3. *$e_m = e'_{n-2}$ and $e'_n = emit(\mathbf{f}, v); e'$ for some expressions $\mathbf{f}$, v, and e'.*

Finally, we need to define $toFOL(\mathbb{L})$ for audit logs $\mathbb{L}$ produced by an instrumented program. Since our audit logs are just sets of formulas of the form $\mathrm{LoggedCall}(t, \mathbf{f}, v)$, we define $toFOL(\mathbb{L}) = \mathbb{L}$.

4.4 Program Rewriting Algorithm

Our program rewriting algorithm $\mathcal{R}_{\Lambda_{\mathrm{call}}}$ takes a Λ_{call} program $\mathfrak{p} = (e, \mathcal{C})$, a logging specification $LS = spec(X_{Guidelines}, \{\mathrm{LoggedCall}\}) \in \mathbf{Calls}$, and produces a $\Lambda_{\log}$ program $\mathfrak{p}' = (e', \mathcal{C}')$ such that e and e' are identical, and $\mathcal{C}'$ is identical to $\mathcal{C}$ except for the addition of $callEvent(\mathbf{h}, v)$ and $emit(\mathbf{h}, v)$ commands. The algorithm is straightforward: we modify the codebase to add $callEvent(\mathbf{h}, v)$ to the definition of any function $\mathbf{h} \in Triggers(LS) \cup \{Logevent(LS)\}$ and add $emit(\mathbf{f}, v)$ to the definition of function $\mathbf{f} = Logevent(LS)$.

Definition 11. *For Λ_{call} program $\mathfrak{p} = (e, \mathcal{C})$ and logging specifications $LS \in \mathbf{Calls}$, define:*

$$\mathcal{R}_{\Lambda_{\mathrm{call}}}((e, \mathcal{C}), LS) = (e, \mathcal{C}')$$

where $\mathcal{C}'(\mathbf{f}) =$

$$\begin{cases} \lambda x.callEvent(\mathbf{f}, x); emit(\mathbf{f}, x); e_{\mathbf{f}} & \text{if } \mathbf{f} = Logevent(LS) \text{ and } \mathcal{C}(\mathbf{f}) = \lambda x.e_{\mathbf{f}} \\ \lambda x.callEvent(\mathbf{f}, x); e_{\mathbf{f}} & \text{if } \mathbf{f} \in Triggers(LS) \text{ and } \mathcal{C}(\mathbf{f}) = \lambda x.e_{\mathbf{f}} \\ \mathcal{C}(\mathbf{f}) & \text{otherwise} \end{cases}$$

This algorithm obeys the required properties, i.e. it is both semantics preserving and sound and complete for a given logging specification.

Theorem 4. *Algorithm $\mathcal{R}_{\Lambda_{\mathrm{call}}}$ is semantics preserving (Definition 4).*

Theorem 5 (Soundness and Completeness). *Algorithm $\mathcal{R}_{\Lambda_{\mathrm{call}}}$ is sound and complete (Definitions 5).*

5 Case Study on a Medical Records System

As a case study, we have developed a tool [2] that enables automatic instrumentation of logging specifications for the OpenMRS system. The implementation is based on the formal model developed in Section 4 which enjoys a correctness guarantee. The logging information is stored in a SQL database consisting of multiple tables, and the correctness of this scheme is established via the monotone mapping defined in Section 3.2. We have also considered how to reduce memory overhead as a central optimization challenge.

OpenMRS is a Java-based open-source web application for medical records, built on the Spring Framework. Previous efforts in auditing for OpenMRS include recording any modification to the database records as part of the OpenMRS core implementation, and logging every function call to a set of predefined records. The latter illustrates the relevance of function invocations as a key factor in logging. Furthermore, function calls define the fundamental unit of "secure operations" in OpenMRS access control [37]. This highlights the relevance of our $\mathbf{Calls}$ logging specification class, particularly as it pertains to specification of break the glass policies, which are sensitive to authorization.

In contrast to previous auditing solutions for OpenMRS, ours allows security administrators to define logging specifications separately from code. Our tool automatically instruments code to correctly support these specifications. This is more convenient, declarative, and less error prone than direct ad hoc instrumentation of code.

System Architecture Summary To clarify the following discussion, we briefly summarize the architecture of our system. Logging specifications are made in the style of $\mathbf{Calls}$ (Definition 9), which can be parsed into JSON objects with a standard form recognized by our system. Instrumentation of legacy code is then accomplished using aspect oriented programming. Parsed specifications are used to identify join points, where the system weaves aspects supporting audit logging into OpenMRS bytecode. These aspects communicate with a proof engine at the joint points to reason about audit log generation, implementing the semantics developed for $\Lambda_{\log}$ in Section 4.3. In our deployment logs are recorded in a SQL database, but our architecture supports other approaches via the use of listeners.

5.1 Break the Glass Policies for OpenMRS

Break the glass policies for auditing are intended to retrospectively manage the same security that is proactively managed by access control (before the glass is broken). Thus it is important that we focus on the same resources in auditing as those focused on by access control. The data model of OpenMRS consists of several domains, e.g. "Patient" and "User" domains contain information about the patients and system users respectively, and the "Encounter" domain includes information regarding the interventions of healthcare providers with patients. In order to access and modify the information in different domains, corresponding service-layer functionalities are defined that are accessible through a web interface. These functionalities provide security sensitive operations through which data assets are handled. Thus, OpenMRS authorization mechanism checks user eligibility to perform these operations [37]. Likewise, we identify these

functionalities in logging specifications, i.e. triggers and logging events are service-layer methods that provide access to data domains, e.g., the patient and user data.

We adapt the logical language of logging specifications developed above (Definition 9), with the minor extension that we allow logging of methods with more than one argument. We note that logging specifications can include other information specified as safe Horn clauses, e.g. ACLs. Here is a simple example of a break the glass auditing policy specified in this form, which states that if the glass is broken by some low-level user, and subsequently patient information is accessed by that user, the access should be logged. The variable `U` refers to the user, and the variable `P` refers to the patient. This specification also defines security levels for two users, `alice` and `admin`. The predicate `@<` defines the usual total ordering on integers.

```
loggedCall(T, getPatient, U, P) :-
  call(T, getPatient, U, P), call(S, breakTheGlass, U),
  @<(S, T), hasSecurityLevel(U, low).

hasSecurityLevel(admin, high).
hassecuritylevel(alice, low).
```

To enable these policies in practice, we have added a "break the glass" button to a user menu in the OpenMRS GUI that can be manually activated on demand. Activation invokes the `breakTheGlass` method parameterized by the user id. We note that breaking the glass does not turn off access control in our current implementation, which we consider a separate engineering concern that is out of scope for this paper.

5.2 Code Instrumentation

To instrument code for log generation, we leverage the Spring Framework that supports aspect-oriented programming (AOP). AOP is used to rewrite code where necessary with "advice", which in our case is *before* certain method invocations (so-called "before advice"). Our advice checks the invoked method names and implements the semantics given in Section 4.3, establishing correctness of audit logging. Join points are automatically extracted from logging specifications, and defined with service-level granularity in a configuration file. Weaving into bytecode is also performed automatically by our system.

For example, in the following excerpt of a configuration file, every interface method of the service `PatientService` is a join point so before invoking each of those methods the advice in `RetroSecurityAdvice` will be woven into the control flow. The `RetroSecurityAdvice` is automatically generated by our system based on the logging specification, but essentially determines whether a method call is a trigger or a logging event and interacts with the proof engine appropriately in each case.

```
<advice>
<point>org.openmrs.api.PatientService</point>
<class>
 org.openmrs.module.retrosecurity.advice.RetroSecurityAdvice
</class>
</advice>
```

Proof Engine According to the the semantics of $\Lambda_{\log}$, it is necessary to perform logical deduction, in particular resolution of $\mathrm{LoggedCall}$ predicates. To this end, we have employed XSB Prolog as a proof engine, due to its reliability and robustness. In order to have a bidirectional communication between the Java application and the engine, InterProlog Java/Prolog SDK [27] is used.

The proof engine is initialized in a separate thread with an interface to the main execution trace. The interface includes methods to define predicates, and to add rules and facts. Asynchrony of the logic engine avoids blocking the "normal" execution trace for audit logging purposes, preserving its original performance. The interface also provides an instant querying mechanism. The instrumented program communicates with the XSB Prolog engine as these interface methods are invoked in advices.

Writing and Storing the Log Asynchronous communication with the proof engine through multi-threading enables us to modularize the deduction of the information that we need to log, separate from the storage and retainment details. This supports a variety of possible approaches to storing log information– e.g., using a strict transactional discipline to ensure writing to critical log, and/or blocking execution until log write occurs. Advice generated by the system for audit log generation just needs to include event listeners to implement the technology of choice for log storage and retainment.

In our application, the logging information is stored in a SQL database consisting of multiple tables. In case new logging information is derived by the proof engine, the corresponding listeners in the main execution trace are notified and the listeners partition and store the logging information in potentially multiple tables. Correctness of this storage technique is established using the monotone mapping rel defined in Section 3.2.

Consider the case where a `loggedCall` is derived by the proof engine given the logging specification in Section 5.1. Here, the instantiation of `U` and `P` are user and patient names, respectively, used in the OpenMRS implementation. However, logged calls are stored in a table called `GetPatL` with attributes `time`, `uid`, and `pid`, where `uid` is the primary key for a `User` table with a `uname` attribute, and `pid` is the primary key for a `Patient` table with a `patient_name` attribute. Thus, for any given logging specification of the appropriate form, the monotonic mapping rel of the following `select` statement gives us the exact information content of the logging specification following execution of an OpenMRS session:

```
select time, "getPatient", uname, patient_name
from GetPatL, User, Patient
where GetPatL.uid = User.uid and GetPatL.pid = Patient.pid
```

5.3 Reducing Memory Overhead

A source of overhead in our system is memory needed to store logging preconditions. We observe that a naive implementation of the intended semantics will add all trigger functions to the logging preconditions, regardless of whether they are redundant in some way. To optimize memory usage, we therefore aim to refrain from adding information about trigger invocations if it is unnecessary for future derivations of audit log information. As a simple example, in the following logging specification it suffices to

add only the first invocation of $\mathbf{g}$ to the set of logging preconditions to infer the relevant logging information.

$$\forall t_0, t_1, x_0, x_1 \,.\, \mathrm{Call}(t_0, \mathbf{f}, x_0) \wedge \mathrm{Call}(t_1, \mathbf{g}, x_1) \wedge t_1 < t_0 \implies \mathrm{LoggedCall}(t_0, \mathbf{f}, x_0).$$

Intuitively, our general approach is to rewrite the body of a given logging specification in a form consisting of different conjuncts, such that the truth valuation of each conjunct is independent of the others. This way, the required information to derive each conjunct is independent of the information required for other conjuncts. Then, if the inference of a $\mathrm{LoggedCall}$ predicate needs a conjunct to be derived only once during the program execution, following derivation of that conjunct, triggers in the conjunct are "turned off", i.e. no longer added to logging preconditions when encountered during execution. Otherwise, the triggers are never turned off. This way, we ensure that none of the invocations of the logging event is missed.

Formally, the logging specification is rewritten in the form

$$\forall t_0, \ldots, t_n, x_0, \ldots, x_n \,.\, \bigwedge_{i=1}^{n} (t_i < t_0) \bigwedge_{k=1}^{L} Q_k \implies \mathrm{LoggedCall}(t_0, \mathbf{g}_0, x_0),$$

where each Q_k is a conjunct of literals with independent truth valuation resting on disjointness of predicated variables. In what follows, a formal description of the technique is given.

Consider the Definition 9. We define Ψ to be the set of all positive literals in the body of $\mathrm{LoggedCall}$ excluding literals $t_i < t_0$ for all $i \in \{1, \cdots, n\}$. Moreover, let's denote the set of free variables of a formula ϕ as $FV(\phi)$, and abuse this notation to represent the set of free variables that exist in a set of formulas. Next, we define the relation $\circledast_{FV}$ over free variables of positive literals in Ψ, which represents whether they are free variables of the same literal, and extend this transitively in the relation $\circledast_{TFV}$.

Definition 12. *Let $\circledast_{FV} \subseteq FV(\Psi) \times FV(\Psi)$ be a relation where $\alpha \circledast_{FV} \beta$ iff there exists some literal $\phi \in \Psi$ such that $\alpha, \beta \in FV(\phi)$. Then, the transitive closure of $\circledast_{FV}$ is denoted by $\circledast_{TFV}$.*

Note that $\circledast_{TFV}$ is an equivalence relation. Let $[\alpha]_{\circledast_{TFV}}$ denote the equivalence class induced by $\circledast_{TFV}$ over $FV(\Psi)$, where $[\alpha]_{\circledast_{TFV}} \triangleq \{\beta \mid \alpha \circledast_{TFV} \beta\}$. Intuitively, each equivalence class $[\alpha]_{\circledast_{TFV}}$ represents a set of free variables in Ψ that are free in a subset of literals of Ψ, transitively. To be explicit about these subsets of literals, we have the following definition (Definition 13). Note that rather than representing an equivalence class using a representative α (i.e., the notation $[\alpha]_{\circledast_{TFV}}$), we may employ an enumeration of these classes and denote each class as C_k, where $k \in 1 \cdots L$. L represents the number of equivalence classes that have partitioned $FV(\Psi)$. In order to map these two notations, we consider a mapping $\omega : FV(\Psi) \to \{1, \cdots, L\}$ where $\omega(\alpha) = k$ if $[\alpha]_{\circledast_{TFV}} = C_k$.

Definition 13. *Let C be an equivalence class induced by $\circledast_{TFV}$. The predicate class $\mathcal{P}_C$ is a subset of literals of Ψ defined as $\mathcal{P}_C \triangleq \{\phi \in \Psi \mid FV(\phi) \subseteq C\}$. We define the independent conjuncts as $Q_C \triangleq \bigwedge_{\phi \in \mathcal{P}_C} \phi$. We also denote $Q_{[\alpha]}$ as Q_k if $\omega(\alpha) = k$. Obviously, $FV(Q_k) = C_k$.*

The above described techniques are used to implement memory overhead mitigation in our OpenMRS retrospective security module– the same mechanism used to perform a `loggedCall` query is used to check whether the independent conjunct Q_C containing a trigger method is satisfiable whenever the trigger is invoked, in which case all triggers in the conjunct are turned off, i.e. no longer added to preconditions when called. In order to prove the correctness of our approach, we have formalized a new calculus $\Lambda'_{\log}$ with memory overhead mitigation capabilities, and shown that the generated log is the same as the log generated in $\Lambda_{\log}$ for the same programs. The reader is referred to our Technical Report [3] for this formalization.

6 Related Work

Previous work by DeYoung et al. has studied audit policy specification for medical (HIPAA) and business (GLBA) processes [20, 19].This work illustrates the effectiveness and generality of a temporal logic foundation for audit policy specification, which is well-founded in a general theory of privacy [18]. Their auditing system has also been implemented in a tool similar to an interactive theorem prover [24]. Their specification language inspired our approach to logging specification semantics. However, this previous work assumes that audit logs are given, and does not consider the correctness of logs. Some work does consider trustworthiness of logs [7], but only in terms of tampering (malleability). In contrast, our work provides formal foundations for the correctness of audit logs, and considers algorithms to automatically instrument programs to generate correct logs.

Other work applies formal methods (including predicate logics [16, 10], process calculi and game theory [28]) to model, specify, and enforce auditing and accountability requirements in distributed systems. In that work, audit logs serve as evidence of resource access rights, an idea also explored in Aura [39] and the APPLE system [22]. In Aura, audit logs record machine-checkable proofs of compliance in the Aura policy language. APPLE proposes a framework based on trust management and audit logic with log generation functionality for a limited set of operations, in order to check user compliance.

In contrast, we provide a formal foundation to support a broad class of logging specifications and relevant correctness conditions. In this respect our proposed system is closely related to PQL [34], which supports program rewriting with instrumentation to answer queries about program execution. From a technical perspective, our approach is also related to trace matching in AspectJ [1], especially in the use of logic to specify trace patterns. However, the concern in that work is aspect pointcut specification, not logging correctness, and their method call patterns are restricted to be regular expressions with no conditions on arguments, whereas the latter is needed for the specifications in $\mathbf{Calls}$.

Logging specifications are related to safety properties [38] and are enforceable by security automata, as we have shown. Hence IRM rewriting techniques could be used to implement them [21]. However, the theory of safety properties does not address correctness of audit logs as we do, and our approach can be viewed as a logging-specific IRM strategy. Guts et al. [25] develop a static technique to guarantee that programs

are properly instrumented to generate audit logs with sufficient evidence for auditing purposes. As in our research, this is accomplished by first defining a formal semantics of auditing. However, they are interested in evidence-based auditing for specific distributed protocols.

Other recent work [23] has proposed log filters as a required improvement to the current logging practices in the industry due to costly resource consumption and the loss of necessary log information among the collected redundant data. This work is purely empirical, not foundational, but provides practical evidence of the relevance of our efforts since logging filters could be defined as logging specifications.

Audit logs can be considered a form of *provenance*: the history of computation and data. Several recent works have considered formal semantics of provenance [9, 8]. Cheney [12] presents a framework for provenance, built on a notion of system traces. Recently, W3C has proposed a data model for provenance, called PROV [5], which enjoys a formal description of its specified constraints and inferences in first-order logic, [13], however the given semantics does not cover the relationship between the provenance record and the actual system behavior. The confidentiality and integrity of provenance information is also a significant concern [26].

7 Conclusion

In this paper we have addressed the problem of audit log correctness. In particular, we have considered how to separate logging specifications from implementations, and how to formally establish that an implementation satisfies a specification. This separation allows security administrators to clearly define logging goals independently from programs, and inspires program rewriting tools that support correct, automatic instrumentation of logging specifications in legacy code.

By leveraging the theory of information algebra, we have defined a semantics of logging specifications as functions from program traces to information. By interpreting audit logs as information, we are then able to establish correctness conditions for audit logs via an information containment relation between log information and logging specification semantics. These conditions allow proof of correctness of program rewriting algorithms that automatically instrument general classes of logging specifications.

We define a particular program rewriting strategy for a core functional calculus that supports instrumentation of logging specifications expressed in first order logic, and then prove this strategy correct. This strategy is then applied to develop a practical tool for instrumenting logging specifications in OpenMRS, a popular medical records system. We discuss implementation features of this tool, including optimizations to minimize memory overhead.

Acknowledgement. This work is supported in part by the National Science Foundation under Grant No. 1408801 and Grant No. 1054172, and by the Air Force Office of Scientific Research.

References

[1] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005. pp. 345–364 (2005)

[2] Amir-Mohammadian, S., Chong, S., Skalka, C.: Retrospective Security Module for OpenMRS. https://github.com/sepehram/retro-security-openmrs (2015)

[3] Amir-Mohammadian, S., Chong, S., Skalka, C.: The theory and practice of correct audit logging. Tech. rep., University of Vermont (October 2015), https://www.uvm.edu/~samirmoh/TR/TR_Audit.pdf

[4] Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. Tech. Rep. TR-649-02, Princeton University (June 2002)

[5] Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., Tilmes, C.: PROV-DM: The PROV data model. http://www.w3.org/TR/2013/REC-prov-dm-20130430 (2013), accessed: 2015-02-07

[6] Biswas, D., Niemi, V.: Transforming privacy policies to auditing specifications. In: HASE 2011. pp. 368–375 (2011)

[7] Böck, B., Huemer, D., Tjoa, A.M.: Towards more trustable log files for digital forensics by means of "trusted computing". In: AINA 2010. pp. 1020–1027. IEEE Computer Society (2010)

[8] Buneman, P., Chapman, A., Cheney, J.: Provenance management in curated databases. In: SIGMOD 2006. pp. 539 – 550 (2006)

[9] Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. Lecture Notes in Mathematics - Springer Verlag pp. 316–330 (2000)

[10] Cederquist, J.G., Corin, R., Dekker, M.A.C., Etalle, S., den Hartog, J.I., Lenzini, G.: Audit-based compliance control. International Journal of Information Security 6(2-3), 133–151 (2007)

[11] Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (And never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1), 146–166 (1989)

[12] Cheney, J.: A formal framework for provenance security. In: CSF 2011. pp. 281–293 (2011)

[13] Cheney, J.: Semantics of the PROV data model. http://www.w3.org/TR/2013/NOTE-prov-sem-20130430 (2013), accessed: 2015-02-07

[14] Chuvakin, A.: Beautiful log handling. In: Oram, A., Viega, J. (eds.) Beautiful security: Leading security experts explain how they think. O'Reilly Media Inc. (2009)

[15] Cook, D., Hartnett, J., Manderson, K., Scanlan, J.: Catching spam before it arrives: Domain specific dynamic blacklists. In: AusGrid 2006. pp. 193–202. Australian Computer Society, Inc. (2006)

[16] Corin, R., Etalle, S., den Hartog, J.I., Lenzini, G., Staicu, I.: A logic for auditing accountability in decentralized systems. In: FAST 2004. pp. 187–201 (2004)

[17] CPMC Press Release: Audit finds employee access to patient files without apparent business or treatment purpose. http://www.cpmc.org/about/press/News2015/phi.html (2015), accessed: 2015-01-30

[18] Datta, A., Blocki, J., Christin, N., DeYoung, H., Garg, D., Jia, L., Kaynar, D.K., Sinha, A.: Understanding and protecting privacy: Formal semantics and principled audit mechanisms. In: ICISS 2011. pp. 1–27 (2011)

[19] DeYoung, H., Garg, D., Jia, L., Kaynar, D., Datta, A.: Privacy policy specification and audit in a fixed-point logic: How to enforce HIPAA, GLBA, and all that. Tech. Rep. CMU-CyLab-10-008, Carnegie Mellon University (April 2010)

[20] DeYoung, H., Garg, D., Jia, L., Kaynar, D.K., Datta, A.: Experiences in the logical specification of the HIPAA and GLBA privacy laws. In: WPES 2010. pp. 73–82 (2010)
[21] Erlingsson, Ú.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University (2003)
[22] Etalle, S., Winsborough, W.H.: A posteriori compliance control. In: SACMAT 2007. pp. 11–20 (2007)
[23] Fu, Q., Zhu, J., Hu, W., Lou, J., Ding, R., Lin, Q., Zhang, D., Xie, T.: Where do developers log? An empirical study on logging practices in industry. In: ICSE 2014. pp. 24–33 (2014)
[24] Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: Theory, implementation and applications. In: CCS 2011. pp. 151–162 (2011)
[25] Guts, N., Fournet, C., Nardelli, F.Z.: Reliable evidence: Auditability by typing. In: ESORICS 2014. pp. 168–183. Springer-Verlag (2009)
[26] Hasan, R., Sion, R., Winslett, M.: The case of the fake Picasso: Preventing history forgery with secures provenance. In: FAST 2009. pp. 1–14 (2009)
[27] InterProlog Consulting: Logic for your app. `http://interprolog.com/` (2014), accessed: 2015-09-27
[28] Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: Towards a theory of accountability and audit. In: ESORICS 2009. pp. 152–167 (2009)
[29] Kemmerer, R.A., Vigna, G.: Intrusion detection: A brief history and overview. Computer 35(4), 27–30 (2002)
[30] King, J.T., Smith, B., Williams, L.: Modifying without a trace: General audit guidelines are inadequate for open-source electronic health record audit mechanisms. In: IHI 2012. pp. 305–314. ACM (2012)
[31] Kohlas, J.: Information Algebras: Generic Structures For Inference. Discrete mathematics and theoretical computer science, Springer (2003)
[32] Kohlas, J., Schmid, J.: An algebraic theory of information: An introduction and survey. Information 5(2), 219–254 (2014)
[33] Lampson, B.W.: Computer security in the real world. IEEE Computer 37(6), 37–46 (2004)
[34] Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: A program query language. In: OOPSLA 2005. pp. 365–383. ACM (2005)
[35] Matthews, P., Gaebel, H.: Break the glass. In: HIE Topic Series. Healthcare Information and Management Systems Society (2009), `http://www.himss.org/files/himssorg/content/files/090909breaktheglass.pdf`
[36] Povey, D.: Optimistic security: A new access control paradigm. In: NSPW 1999. pp. 40–45 (1999)
[37] Rizvi, S.Z., Fong, P.W.L., Crampton, J., Sellwood, J.: Relationship-based access control for an open-source medical records system. In: SACMAT 2015. pp. 113–124 (2015)
[38] Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)
[39] Vaughan, J.A., Jia, L., Mazurak, K., Zdancewic, S.: Evidence-based audit. In: CSF 2008. pp. 177–191 (2008)
[40] Weitzner, D.J.: Beyond secrecy: New privacy protection strategies for open information spaces. IEEE Internet Computing 11(5), 94–96 (2007)
[41] Weitzner, D.J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J.A., Sussman, G.J.: Information accountability. Communications of the ACM 51(6), 82–87 (2008)
[42] Zhang, W., Chen, Y., Cybulski, T., Fabbri, D., Gunter, C.A., Lawlor, P., Liebovitz, D.M., Malin, B.: Decide now or decide later? Quantifying the tradeoff between prospective and retrospective access decisions. In: CCS 2014. pp. 1182–1192 (2014)
[43] Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: Simultaneous identification of multiple bugs. In: ICML 2006. pp. 1105–1112. ACM (2006)