The Theory and Practice of Correct Audit Logging

Sepehr Amir-Mohammadian Stephen Chong Christian Skalka

October 20, 2015

Abstract

Auditing has become increasingly important to the theory and practice of cyber security. 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 auditing techniques are formal, since the relation of the log itself to program execution is unclear. This work 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. Logging semantics is sufficiently general, so that the guarantees extend to various approaches of audit logging. As a case study, we demonstrate the incorporation of the proposed techniques and features in healthcare informatics. In particular, we consider auditing for break the glass policies, wherein authorization is replaced by auditing in emergency conditions.

Contents

List of Figures

Chapter 1

Introduction

Retrospective security is the enforcement of security, or detection of security violations, after program execution [1, 2, 3]. 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 transactions. Another example is a hospital whose employees are trusted to access confidential patient records, but who might (rarely) violate this trust [4]. Upon detection of such violations, security is enforced retrospectively by holding responsible employees accountable [5].

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) [6]. 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 [5].

Auditing underlies retrospective security frameworks and has become increasingly important to the theory and practice of cyber security. By recording appropriate aspects of a computer system's execution an audit log (and subsequent examination of the audit log) can enable detection of violations, and 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 [7]. 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 [8, 9, 10], and is a critical factor in the design of tools that generate and employ audit logs (e.g., spam filters [11]).

A main goal of this work is to establish a formal foundation for audit logging, especially to establish general correctness conditions for audit logs. We define a general semantics of audit logs using the theory of *information algebra* [12]. We interpret both program execution traces and audit logs as information elements in an information algebra. 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 supports clearer definitions and more direct reasoning. Additionally, it enables algorithms for implementing general classes of logging specifications. Our formal theory establishes conditions that guarantee enforcement of logging specifications by such algorithms. As we will show, correct instrumentation of logging specifications is a safety property, hence enforceable by security automata [13]. Inspired by related approaches to security automata implementation [14], 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.

This approach would allow system administrators to define logging specifications which are automatically instrumented in code, including legacy code. Implementation details and matters such as optimization can be handled by the general program rewriting algorithm, not the logging specification. Furthermore, establishing correctness of a program rewriting algorithm provides an important security guarantee. Such an algorithm ensures that logging specifications will be implemented correctly, even if the rewritten source code contains malicious code or programmer errors.

We consider a case study of our approach, a program rewriting algorithm for correct instrumentation of logging specifications in OpenMRS [15], 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 Summary and Main Technical Results

In the following we informally introduce concepts and relevant terminology, discuss a motivating example, and identify our threat model. In Chapter 2 we define a semantics of auditing, and establish conditions for correctness of audit rewriting algorithms. That is, we define what it means for a program instrumentation to correctly log information. In Chapter 3, we study the instantiations of the auditing semantics and transformation of these instantiations to each other. In Chapter 4, we consider a particular class of logging specifications, present a rewriting algorithm to support this class, and prove that this algorithm is correct. In Chapter 5, we discuss a case study on health informatics, particularly OpenMRS system. We conclude with the related work and the summary remarks in Chapter 6.

The main technical contributions of this work are as follows. In Chapter 2, we characterize logging specifications and correctness conditions for audit logs, in a high-level manner using information algebra. In particular, we obtain formal notions of soundness and completeness of program rewriting for auditing (Definitions 2.5.2 and 2.5.3). In Chapter 3, we formulate information algebras based on first-order logic (FOL) and relational algebra that are shown to satisfy necessary conditions (Theorem 3.1.1 and Corollary 3.1.1) to enjoy information-algebraic properties, including a partial information order. We then leverage FOL-based information algebra to define a formal semantics of program auditing (Definition 3.1.5). In Chapter 4, we prove that a rewriting algorithm is sound and complete with respect to a specific class of logging specifications (Theorem 4.4.2). This illustrates how our auditing semantics can be leveraged to prove program instrumentation correctness for particular rewriting algorithms. In Chapter 5, we discuss the deployment of correct audit logging mechanism for OpenMRS system, and

propose techniques to reduce memory overhead.

1.2 Background and Terminology

In this Section we aim to establish terminology and concepts, both for use in the report and to clarify under-appreciated details of auditing. We first summarize traditional goals of auditing, in order to categorize typical application spaces.

Audit Logs as Evidence Many formal approaches to auditing consider how audit logs can provide evidence for post-facto justification of resource access [16]. This evidence is usually in the form of a proof of policy compliance, or supporting facts for such a proof. In such settings, audit logs also include information to support accountability, e.g., to allow analyses to determine who is responsible for policy violations.

Retrospective Dynamic Analysis Since an audit log provides a partial record of program execution, it can be used to retrospectively analyze program execution, to detect security violations, or to find performance bottlenecks. The ability to perform retrospective dynamic analysis is especially important when static analyses are undecidable (e.g., [17]), or impossible (e.g., due to application-specific use of a framework [18]).

Surveillance and Accountability Surveillance is a prevalent security technique, where potentially insecure actions are allowed but events are recorded for subsequent analysis. As a real-world example, in a medical informatics scenario certain users may be given access to sensitive patient information in case of an emergency, but users should be held accountable post-facto in the event of a security violation (known as “break the glass policies”) [7]. Such mechanisms have been identified as part of the solution for privacy and mobile security (aka mhealth security) [19, 20].

Surveillance and accountability is supported by auditing, and leveraged to enhance existing security models, e.g., access control and information flow models. Other authors have termed this approach "optimistic security" [2].

To avoid confusion in our formal presentation and discussion we define some terminology. These definitions are also intended to better isolate and describe elements of auditing, in particular we note that auditing of processes typically involves two distinct policies: one for generating the audit log itself, and one for analyzing the audit log.

We use the following terminology throughout the report.

An *execution trace* is a complete record of program execution in a formal operational semantics.

An *audit log* is a record of program execution. The format of audit logs varies, but essentially comprises the information derived from the full execution trace. A *log query* is any question that can be asked of the audit log, though it is typically said that the program execution is "being audited" in this case. This reveals an implicit understanding that audit logs bear a knowable relation with processes.

A *logging specification* specifies how audit logs should be generated, that is, what would be the relation between the information contained in the execution trace and in the audit log. Logging specifications should typically ensure that audit logs contain enough information, and in an appropriate form, to answer a given log query (or set of log queries). A logging specification is *enforced* by a program if execution of the program produces an appropriate audit log. In this work we are interested in formally defining logging specifications, and automatically enforcing them. A *rewriting algorithm* takes a logging specification and a program, and instruments the program to enforce the logging specification. A rewriting algorithm may support only a limited class of logging specifications.

An *auditing policy* is the combination of a log query and a logging specification. Thus,

an auditing policy describes both a logging specification to enforce, and a query to ask of the resulting audit logs.

1.3 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 include break the glass policies used in electronic medical record systems [21]. 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 [7]. 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 work. Other work has focused on formalisms for querying logs [16, 22], however these works presuppose correctness of audit logs for true accountability.

1.4 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. Non-malleability of generated audit logs, while important, is beyond the scope of this work.

Chapter 2

A Semantics of Audit Logging

Our goal in this Chapter 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* [23, 12], 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.

2.1 Introduction to Information Algebra

Information algebra is the algebraic study of the theory of information. In information algebra, information is seen as a collection of separate elements. Each information element can be queried for further refinement and also aggregated with other information elements. To this end, the algebra consists of two domains: an information domain and a query domain. The information domain Φ is the set of information elements that can be aggregated in order to build more inclusive information elements. The query domain E is a lattice of querying sublanguages

in which the partial order relation among these sublanguages represents the granularity of the queries. In order to aggregate and query the information elements, the following operations are defined.

Definition 2.1.1 *Any information algebra* (Φ, E) *includes two basic operators:*

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

The two-sorted algebra (Φ, E) is an information algebra if the combination and focusing operations defined in Definition 2.1.1 meet specific properties.

Definition 2.1.2 *Any two-sorted algebra* (Φ, E) *with operators* $\otimes : \Phi \times \Phi \rightarrow \Phi$ *and* $\Rightarrow: \Phi \times E \rightarrow \Phi$ *is an information algerba iff the following properties hold:*

- Φ *is a semigroup under combination, i.e., associativity and commutativity hold for* $\otimes$ *and there exists a neutral element* $I \in \Phi$*,*
- *Transitivity of focusing:* $(X^{\Rightarrow L})^{\Rightarrow M} = X^{\Rightarrow L \cap M}$ *for all* $X \in \Phi$ *and* $L, M \in E$*,*
- *Combination:* $(X^{\Rightarrow L} \otimes Y)^{\Rightarrow L} = X^{\Rightarrow L} \otimes Y^{\Rightarrow L}$ *for all* $X, Y \in \Phi$ *and* $L \in E$*,*
- *Support: For all* $X \in \Phi$*, there exists some* $L \in E$ *such that* $X^{\Rightarrow L} = X$*, and*
- *Idempotence:* $X \otimes X^{\Rightarrow L} = X$ *for all* $X \in \Phi$ *and* $L \in E$.

Using the combination operator we can define a partial order relation on Φ to compare the information contained in the elements of Φ. 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.

Definition 2.1.3 *X is contained in Y, denoted as $X \leq Y$, for all $X, Y \in \Phi$ iff $X \otimes Y = Y$.*

Definition 2.1.4 *We say that X and Y are* information equivalent*, and write $X = Y$, iff $X \leq Y$ and $Y \leq X$.*

For a more detailed account of information algebra, the reader is referred to a definitive survey paper [23].

2.2 Logging Specifications

Following [13], 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 Chapter 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.

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 Chapter 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 Chapter 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 [13].

Definition 2.2.1 *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 [24], 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.

2.3 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.

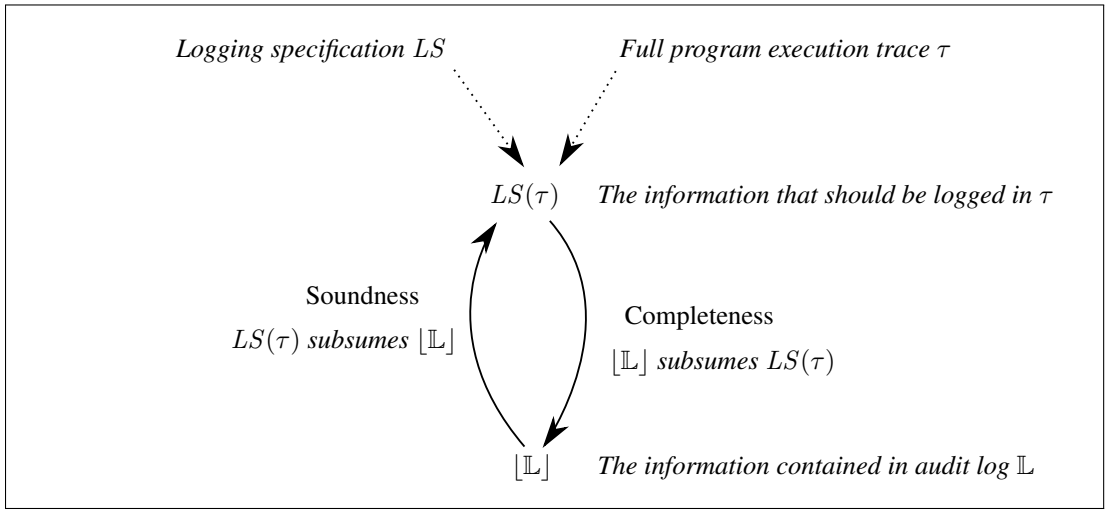

Figure 2.1: Concept diagram: Logging specification and correctness of audit logs.

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

Definition 2.3.2 *Audit log* $\mathbb{L}$ *is* complete with respect to logging specification LS and execution trace τ *iff* $LS(\tau) \leq \lfloor \mathbb{L} \rfloor$.

Figure 2.1 illustrates graphically the relations of soundness and completeness of audit logs with respect to the semantics of logging.

The relation to log queries. As discussed in Section 1.3, 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 [25].

2.4 Correct Logging Instrumentation is a Safety Property

In case program executions generate audit logs, we write $\tau \rightsquigarrow \mathbb{L}$ to mean that trace τ generates $\mathbb{L}$, i.e. $\tau = \kappa_0 \ldots \kappa_n$ and $logof(\kappa_n) = \mathbb{L}$ where $logof(\kappa)$ 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 2.4.1 *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 2.2.1, implies that ideal instrumentation of any logging specification is a safety property in the sense defined by Schneider [13].

Theorem 2.4.1 *For all logging specifications LS, the set of ideally instrumented traces is a safety property.*

Proof. If τ is ideally instrumented for LS, then it is prefix-closed by definition. Furthermore, if τ is not ideally instrumented for LS, then it will definitely be rejected in a finite amount of time, since any information in $LS(\tau)$ is encountered after execution of a finite prefix σ of τ by Definition 2.2.1. These two facts obtain the result. □

This result implies that e.g. edit automata can be used to enforce instrumentation of logging specifications (see Section 4.5). However, theory related to safety properties and their enforcement by execution monitors [13, 26] do not provide an adequate semantic foundation for audit log generation, nor an account of soundness and completeness of audit logs.

2.5 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. 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.[1] 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. Possi-

[1] 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.

ble definitions are that traces produce the same final value, or that traces when restricted to a set of memory locations are equivalent up to stuttering. We provide an explicit definition of correspondence for λ-calculus source and target languages in Chapter 4.

Definition 2.5.1 *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., $\tau :\approx \tau'$), 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 Chapter 4, Definition 4.3.1). 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 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 (i) of Definition 2.5.1, not condition (ii), so semantics preservation is an independent condition.

Definition 2.5.2 *Rewriting algorithm $\mathcal{R}$ is* sound *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 with respect to LS and τ.

Definition 2.5.3 *Rewriting algorithm $\mathcal{R}$ is* 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 complete with respect to LS and τ.*

Chapter 3

Languages for Logging Specifications

In this chapter, we go into more detail about information algebra and why it is a good foundation for logging specifications and semantics.

3.1 Support for Various Approaches

Various approaches are taken to audit log generation and representation, including logical [22], database [27], and probabilistic approaches [28]. 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.

3.1.1 First Order Logic (FOL)

Logics have been used in several well-developed auditing systems [29, 30], 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.

The properties given in the following Lemma are stated without proof since they are self-evident properties of FOL deduction.

Lemma 3.1.1 *Each of the following properties hold:*

1. $X \vdash \phi$ *for each* $\phi \in X$
2. *If* $X \vdash \phi$ *for each* ϕ *in* Y *and* $Y \vdash \psi$*, then* $X \vdash \psi$

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 [12], however our definition of closure is syntactic, not semantic.

Definition 3.1.1 *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.*

Due to the definition of preconditions, we will be particularly interested in proving properties of sets $C_L(X)$:

Definition 3.1.2 *For each sublanguage* $L \in \mathcal{S}$*, we define closure operator* $C_L(X)$*:*

$$C_L(X) = C(X) \cap L.$$

An important point about such sets is that their closures contain tautological assertions, which may involve predicates P which are not included in L. However, in tautological assertions any predicate does as well as any other, which is an important fact to get hold of for our proofs. Thus we will identify a "dummy" predicate that, in essence, allows us to treat a canonical form of tautologies.

Definition 3.1.3 *We reserve a unary* dummy *predicate D with a countably infinite domain of constants* $\mathbf{c}$*, and posit an injective function from distinct concrete assertions $P(\bar{\mathbf{c}})$ to distinct $D(\mathbf{c})$. We further define $norm_L(\phi)$ to be the formula ϕ' which is the same as ϕ, but where each $P(\bar{\mathbf{c}}) \notin L$ is replaced with its corresponding image $D(\mathbf{c})$ in the injection. The pointwise extension of $norm_L$ to sets X is denoted $norm_L(X)$.*

Now, we demonstrate canonical forms:

Lemma 3.1.2 *$X \cap L \cup Y \vdash \phi$ iff $X \cap L \cup norm_L(Y) \vdash norm_L(\phi)$.*

Proof. First we prove the left-to-right implication by induction on the derivation of $C(X) \cap L \cup Y \vdash \phi$ and case analysis on the last step in the derivation.

Case Axiom. In this case $\phi \in X \cap L \cup Y$, so either $\phi \in Y$, or $\phi \in X \cap L$. In the former subcase, $norm_L(\phi) \in norm_L(Y)$ by definition and $norm_L(\phi) = \phi$ in the latter subcase. In either subcase, the result holds axiomatically.

Case $\rightarrow$ elimination. In this case we have:

$$\frac{X \cap L \cup Y \vdash \psi \rightarrow \phi \qquad X \cap L \cup Y \vdash \psi}{X \cap L \cup Y \vdash \phi}$$

By the induction hypothesis we have:

$$X \cap L \cup norm_L(Y) \vdash norm_L(\psi \rightarrow \phi) \qquad X \cap L \cup norm_L(Y) \vdash norm_L(\psi)$$

But $norm_L(\psi \to \phi) = norm_L(\psi) \to norm_L(\psi)$ by definition, so the result follows in this case by modus ponens. □

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.

Lemma 3.1.2 shows that wlog we can modify $\mathcal{S}$ so that every $L \in \mathcal{S}$ contains D. Hence we have immediately:

Lemma 3.1.3 *For all $L, M \in \mathcal{S}$ and $\phi \in \mathcal{F}$, $norm_L(\phi) \in L$, and if $\phi \in M$ then $norm_L(\phi) \in M$.*

This allows us to prove the following important auxiliary results about closures.

Lemma 3.1.4 *If $C(X) \cap L \vdash \phi$ and $\phi \in M$, then $C(X) \cap L \cap M \vdash \phi$.*

Proof. By Lemma 3.1.3 we have $norm_L(\phi) \in L$, and by Lemma 3.1.2 we have $C(X) \cap L \vdash norm_L(\phi)$, hence $norm_L(\phi) \in C(X) \cap L$. But also $norm_L(\phi) \in M$ by assumption and Lemma 3.1.3, so also $norm_L(\phi) \in C(X) \cap L \cap M$. Hence $C(X) \cap L \cap M \vdash norm_L(\phi)$ as an axiom, therefore the result follows by Lemma 3.1.2. □

Lemma 3.1.5 *If $C_L(C_M(X)) \cup Y \vdash \phi$ then $C_{M \cap L}(X) \cup Y \vdash \phi$.*

Proof. The result follows by induction on $C_L(C_M(X)) \cup Y \vdash \phi$ and case analysis on the last step in the derivation. Most cases follow in a straightforward manner; the presence of Y in the formulation is to allow for additional hypotheses, as for example in the case of $\to$ introduction, as follows.

Case $\rightarrow$ introduction. In this case $\phi = \phi_1 \rightarrow \phi_2$, and we have:

$$\frac{C_L(C_M(X)) \cup Y \cup \{\phi_1\} \vdash \phi_2}{C_L(C_M(X)) \cup Y \vdash \phi_1 \rightarrow \phi_2}$$

but then by the induction hypothesis the judgement $C_{M \cap L}(X) \cup Y \cup \{\phi_1\} \vdash \phi$ is derivable, so the result follows by $\rightarrow$ introduction.

The interesting case is the axiomatic one, i.e. where $\phi \in C_L(C_M(X)) \cup Y$, and specifically the subcase where $\phi \in C_L(C_M(X))$, which follows by Lemma 3.1.4. □

The following Lemma completes the necessary preconditions to prove that the construction Φ_{FOL} is a "domain-free" information algebra [12].

Lemma 3.1.6 *Each of the following properties hold:*

1. *If $X \subseteq Y$ then $C(X) \subseteq C(Y)$.*
2. $C(X \cup Y) = C(X \cup C(Y))$
3. $C(C_L(C_M(X))) = C(C_{M \cap L}(X))$
   for $X \subseteq \mathcal{F}$ and $L, M \in \mathcal{S}$
4. $C_L(C_L(X) \cup Y) = C_L(C_L(X) \cup C_L(Y))$

Proof. Properties (1) and (2) are a consequence of Lemma 3.1.1 as demostrated in [12].

Proof of (3). By definition:

$$C(C_L(C_M(X))) = C(C(C(X) \cap M) \cap L) \qquad C(C_{M \cap L}(X)) = C(C(X) \cap M \cap L)$$

Since $C(X) \cap M \subseteq C(C(X) \cap M)$ by Lemma 3.1.1 property (1), therefore by property (1) in

the current Lemma:

$$C(C(X) \cap M \cap L) \subseteq C(C(C(X) \cap M) \cap L).$$

It thus remains to show that:

$$C(C(C(X) \cap M) \cap L) \subseteq C(C(X) \cap M \cap L)$$

which follows by definition of closure and an application of Lemma 3.1.5, taking $Y = \varnothing$ in that Lemma.

Proof of (4). By (2), we have:

$$C(C_L(X) \cup Y) = C(C_L(X) \cup C(Y))$$

and thus also:

$$C(C_L(X) \cup (C(Y) \cap L)) \subseteq C(C_L(X) \cup C(Y))$$

which establishes:

$$C_L(C_L(X) \cup C_L(Y)) \subseteq C_L(C_L(X) \cup Y).$$

For brevity in the remaining, define:

$$A = C_L(C_L(X) \cup C(Y)) \qquad B = C_L(C_L(X) \cup C_L(Y)).$$

To prove the result it now suffices to establish that $A \subseteq B$, so we assume on the contrary that there exists some $\phi \in L$ where $A \vdash \phi$ but $B \nvdash \phi$. Now, clearly $\phi \notin C(X)$ and $\phi \notin C(Y)$, since in these cases it must be that $B \vdash \phi$ holds. Therefore there exist some minimal nonempty subsets $C \subseteq C(Y)$ and $D \subseteq C_L(X)$ such that $C \cup D \vdash \phi$. Let ψ_D be the conjunction of

terms in D. Clearly $\psi_D \in L$. Furthermore, by properties of logic, $C \vdash \psi_D \rightarrow \phi$ holds, so that $\psi_D \rightarrow \phi \in C(Y)$, and since $\psi_D \in L$ and $\phi \in L$ by construction and assumption, therefore $\psi_D \rightarrow \phi \in L$, hence $\psi_D \rightarrow \phi \in C_L(Y)$. But $\psi_D \in C_L(X)$ necesarily, so $C_L(X) \cup C_L(Y) \vdash \phi$ by modus ponens and (1). Thus $B \vdash \phi$ also by (1), which is a contradiction, etc. □

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 3.1.4 *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 an information algebra. FOL has been treated as an information algebra before, but our definitions of combination and focusing and hence the result are novel.

Theorem 3.1.1 *Structure* $(\Phi_{FOL}, \mathcal{S})$ *with focus operation* $X^{\Rightarrow S}$ *and combination operation* $X \otimes Y$ *forms a* domain-free information algebra.

Proof. The following properties hold immediately according to Lemma 3.1.6 and Lemma 3.1.1, and thus $(\Phi_{FOL}, \mathcal{S})$ is an information algebra [12]:

- *Semigroup*: Φ is associative and commutative under combination, and $C(\varnothing)$ is a neutral element with $X \otimes C(\varnothing) = X$ for all $X \in \Phi$.

- *Transitivity*: $(X^{\Rightarrow L})^{\Rightarrow M} = X^{\Rightarrow L \cap M}$ for all $X \in \Phi$ and $L, M \in \mathcal{S}$.
- *Combination*: $(X^{\Rightarrow L} \otimes Y)^{\Rightarrow L} = X^{\Rightarrow L} \otimes Y^{\Rightarrow L}$ for all $X, Y \in \Phi$ and $L \in \mathcal{S}$.
- *Support*: $X^{\Rightarrow \mathcal{F}} = X$ for all $X \in \Phi$.
- *Idempotence*: $X \otimes X^{\Rightarrow L} = X$ for all $X \in \Phi$ and $L \in \mathcal{S}$.

□

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 3.1.1 *We return to the example described in Section 1.3 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* 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 3.1.5 *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 $spec(\psi_H, \{\mathrm{LoggedCall}\})$.

3.1.2 Relational Database

Relational algebra is a canonical example of an information algebra. We define databases D as sets of relations, where a relation X is a set of *tuples* f. 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. Focusing is the restriction to particular relations in a database, and combination is 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}. In this context, a trace can be interpreted as a collection of relations, and logging specifications can be defined using selects. Relational databases are also heavily used for storing and querying audit logs.

Let $\mathcal{A}$ be a denumerable set of attribute names. Moreover, let $\mathbb{R}$ be the universe for relations, i.e., $\mathbb{R} = \{R \subseteq A_{a_1} \times \cdots A_{a_m} \mid a_i \in \mathcal{A}\}$. Note that A_{a_i} is the domains of values for attribute a_i. We denote the arity of a relation R with $arity(R)$.

Definition 3.1.6 *Let $Name : \mathbb{R} \to \mathcal{P}(\mathcal{A})$ be defined as $Name(R) = \{a_1, \cdots, a_{arity(R)}\}$, if $R \subseteq A_{a_1} \times \cdots A_{a_{arity(R)}}$.*

Definition 3.1.7 *Let database D be a finite subset of $\mathbb{R}$ containing finite relations, i.e., a database D is a finite collection of relations $R \in \mathbb{R}$, where each R is a finite set of tuples defining the relation. Φ_{RA} is defined as the set of all databases.*

We also define the querying sublanguages as the sets of relation names, i.e., $S \in \mathcal{P}(\mathcal{P}(\mathcal{A}))$. Next, we define the information algebra operations:

Definition 3.1.8 *Define:*

- *Focusing: $D^{\Rightarrow S} = \{R \in D \mid Name(R) \in S\}$, where $D \in \Phi_{RA}$, and $S \in \mathcal{P}(\mathcal{P}(\mathcal{A}))$ and finite,*
- *Combination: $D_1 \otimes D_2 = \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1, 2\}, Name(R_1) = Name(R_2)\}$.*

Note that in case some relation is not defined in a database, we assume it is defined as an empty relation. We also define a mapping which represents non-trivial relation names in a database:

Definition 3.1.9 *Let* $Names : \Phi_{RA} \to \mathcal{P}(\mathcal{P}(\mathcal{A}))$ *be defined as*

$$Names(D) = \{Name(R) \mid R \in D, R \neq \emptyset\}.$$

In what follows we show that $(\Phi_{RA}, \mathcal{P}(\mathcal{P}(\mathcal{A})))$ is an information algebra.

Lemma 3.1.7 Φ_{RA} *is a semigroup.*

Proof. We need to show that

- Φ_{RA} is associative on combination:

  Holds straightforwardly based on the associativity of union on sets:

$$\begin{aligned} D_1 \otimes (D_2 \otimes D_3) &= D_1 \otimes \{R_2 \cup R_3 \mid R_i \in D_i, i \in \{2,3\}, Name(R_2) = Name(R_3)\} \\ &= \{R_1 \cup (R_2 \cup R_3) \mid R_i \in D_i, i \in \{1,2,3\}, \\ &\qquad Name(R_1) = Name(R_2) = Name(R_3)\} \\ &= \{(R_1 \cup R_2) \cup R_3 \mid R_i \in D_i, i \in \{1,2,3\}, \\ &\qquad Name(R_1) = Name(R_2) = Name(R_3)\} \\ &= \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\} \otimes D_3 \\ &= (D_1 \otimes D_2) \otimes D_3. \end{aligned}$$

- Φ_{RA} is commutative on combination:

  Holds straightforwardly based on the commutativity of union on sets:

$$\begin{aligned} D_1 \otimes D_2 &= \{R_1 \cup R_2 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\} \\ &= \{R_2 \cup R_1 \mid R_i \in D_i, i \in \{1,2\}, Name(R_1) = Name(R_2)\} \\ &= D_2 \otimes D_1. \end{aligned}$$

- There exists a neutral element I such that for all D, $D \otimes I = D$:

  Let $I = \{\emptyset\}$. Obviously, $D \otimes I = D$ as $\emptyset$ is the neutral element for union.

□

Lemma 3.1.8 *Transitivity:* $(D^{\Rightarrow S_1})^{\Rightarrow S_2} = D^{\Rightarrow S_1 \cap S_2}$.

Proof.

$$\begin{aligned}(D^{\Rightarrow S_1})^{\Rightarrow S_2} &= \{R \in D \mid Name(R) \in S_1\}^{\Rightarrow S_2} \\ &= \{R \in D \mid Name(R) \in S_1, Name(R) \in S_2\} \\ &= \{R \in D \mid Name(R) \in S_1 \cap S_2\} \\ &= D^{\Rightarrow S_1 \cap S_2}.\end{aligned}$$

□

Lemma 3.1.9 *Combination:* $({D_1}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = {D_1}^{\Rightarrow S} \otimes {D_2}^{\Rightarrow S}$.

Proof.

$$\begin{aligned}({D_1}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} &= (\{R_1 \in D \mid Name(R) \in S\} \otimes D_2)^{\Rightarrow S} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, Name(R_1) \in S, R_2 \in D_2, \\ &\qquad Name(R_1) = Name(R_2)\}^{\Rightarrow S} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, Name(R_1) \in S, R_2 \in D_2, \\ &\qquad Name(R_1) = Name(R_2), Name(R_1 \cup R_2) \in S\}.\end{aligned}$$

Obviously, if $Name(R_1) = Name(R_2)$, then $Name(R_1 \cup R_2) = Name(R_1)$. We thus have

$$(D_1{}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = \{R_1 \cup R_2 \mid R_1 \in D_1, Name(R_1) \in S, R_2 \in D_2, \\ Name(R_1) = Name(R_2)\}$$

Moreover,

$$\begin{aligned} D_1{}^{\Rightarrow S} \otimes D_2{}^{\Rightarrow S} &= \{R_1 \in D \mid Name(R_1) \in S\} \otimes \{R_2 \in D \mid Name(R_2) \in S\} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, Name(R_1) \in S, R_2 \in D_2, Name(R_2) \in S, \\ &\qquad Name(R_1) = Name(R_2)\} \\ &= \{R_1 \cup R_2 \mid R_1 \in D_1, Name(R_1) \in S, R_2 \in D_2, \\ &\qquad Name(R_1) = Name(R_2)\}. \end{aligned}$$

Thus, $(D_1{}^{\Rightarrow S} \otimes D_2)^{\Rightarrow S} = D_1{}^{\Rightarrow S} \otimes D_2{}^{\Rightarrow S}$. □

Lemma 3.1.10 *Support:* $\forall D, \exists S, D^{\Rightarrow S} = D$.

Proof. Let $S = Names(D)$. Then, $D^{\Rightarrow Names(D)} = D$. □

Lemma 3.1.11 *Idempotence:* $D \otimes D^{\Rightarrow S} = D$.

Proof.

$$\begin{aligned} D \otimes D^{\Rightarrow S} &= \{R \mid R \in D\} \otimes \{R \in D \mid Name(R) \in S\} \\ &= \{R \in D \mid Name(R) \notin S\} \cup \{R \in D \mid Name(R) \in S\} \\ &= D. \end{aligned}$$

□

Corollary 3.1.1 $(\Phi_{RA}, \mathcal{P}(\mathcal{P}(\mathcal{A})))$ *is an information algebra.*

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 crucial 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 another 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* [31], 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:

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

Obviously, $Name(R_{\mathrm{P}}(X)) = \{attrs(\mathrm{P})(1), \cdots, attrs(\mathrm{P})(n)\}$, for n-ary predicate symbol P.

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 3.2.1 *rel is a monotone mapping.*

Proof. We need to show that $X \leq_{FOL} Y$ implies $rel(X) \leq_{RA} rel(Y)$.
From $X \leq_{FOL} Y$ we have $\mathfrak{H}(X) \subseteq \mathfrak{H}(Y)$. Let $R \in rel(X)$ and $R' \in rel(Y)$, such that $Name(R) = Name(R')$. Then, $R = R_{\mathrm{P}}(X)$ and $R' = R_{\mathrm{P}}(Y)$, for some n-ary predicate symbol P such that $Name(R) = \{attrs(\mathrm{P})(1), \cdots, attrs(\mathrm{P})(n)\}$. Since $\mathfrak{H}(X) \subseteq \mathfrak{H}(Y)$,

$$\begin{aligned} R &= \{tuple(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(X)\} \\ &\subseteq \{tuple(\mathrm{P}(x_1, \ldots, x_n)) \mid \mathrm{P}(x_1, \ldots, x_n) \in \mathfrak{H}(Y)\} = R'. \end{aligned}$$

Therefore, $R \cup R' = R'$. Then,

$$\begin{aligned} rel(X) \otimes rel(Y) &= \{R \cup R' \mid R \in rel(X), R' \in rel(Y), Name(R) = Name(R')\} \\ &= \{R' \mid R \in rel(X), R' \in rel(Y), Name(R) = Name(R')\} \\ &= rel(Y). \end{aligned}$$

This implies $rel(X) \leq_{RA} rel(Y)$ by information containment definition. □

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 $\mathbf{R}$ 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.

Chapter 4

Rewriting Programs with Logging Specifications

Since correct logging instrumentation is a safety property (2.4), there are various implementation strategies. For example, one could define an edit automata that enforces the property (see Section 4.5), that could be implemented either as a separate program monitor or using IRM techniques [14]. 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 a prototypical case study. 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 [32]), 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.

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

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

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

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

An execution trace τ is a sequence of configurations, and for a 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 $\kappa_0 = (e, 0, \mathcal{C})$ 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 $\langle \mathcal{C} \rangle_n = \bigcup_{\mathbf{f} \in \text{dom}(\mathcal{C})} \{\text{Codebase}(n, \mathbf{f}, \mathcal{C}(\mathbf{f}))\}$[1].

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

We define $toFOL(\tau)$ for a potentially infinite execution trace $\tau = \kappa_0 \kappa_1 \ldots$ by defining it over

[1] 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.

its prefixes. Let $\mathrm{prefix}(\tau)$ denote the set of prefixes of τ. Then,

$$toFOL(\tau) = \bigcup_{\sigma \in \mathrm{prefix}(\tau)} toFOL(\sigma),$$

where $toFOL(\sigma) = toFOL(\kappa_0) \cup \cdots \cup toFOL(\kappa_n)$, for $\sigma = \kappa_0 \ldots \kappa_n$. 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 [31]. Specifications in **Calls** support logging of calls to a specific function $\mathbf{f}$ that happen after functions $\mathbf{g}_1, \ldots, \mathbf{g}_n$ 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 4.2.1 **Calls** *is the set of all logging specifications* $spec(X, \{\mathrm{LoggedCall}\})$ *where* X *contains a safe Horn clause of the following form:*

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

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 2.2.1, 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 Λ_{log} extends Λ_{call} syntax with a command to track logging preconditions ($callEvent(\mathbf{f}, v)$) 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 $\mathbb{L}$.

$$e ::= \ldots \mid callEvent(\mathbf{f}, v); e \mid emit(\mathbf{f}, v); e \qquad \textit{expressions}$$

$$\kappa ::= (e, X, n, \mathbb{L}, \mathcal{C}) \qquad \textit{configurations}$$

The semantics of Λ_{log} extends the semantics of Λ_{call} with new rules for $callEvent(\mathbf{f}, v)$ and $emit(\mathbf{f}, v)$, which update the set of logging preconditions and audit log respectively.

Precondition

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

Log

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

NoLog

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

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.

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. Also, since events are intended to flag functions called in the immediately preceding timestep, as we will eventually specify for program rewriting, timesteps are decremented for triggers and logged calls.

The rules Log and NoLog rely on checking whether $X_{Guidelines}$ and logging preconditions X entail $\mathrm{LoggedCall}(n-1, \mathbf{f}, v)$. This can be accomplished using off-the-shelf theorem provers for Horn clause logics, such as Datalog or Prolog.

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 $\kappa_{i-1} \rightarrow \kappa_i$.

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 4.3.1 *Given source language execution trace $\tau = \kappa_0 \ldots \kappa_m$ and target language execution trace $\tau' = \kappa'_0 \ldots \kappa'_n$, 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 Λ_{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 4.4.1 *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 $\mathcal{C}'(\mathbf{f}) =$

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

Program rewriting algorithm $\mathcal{R}_{\Lambda_{\mathrm{call}}}$ is semantics preserving, sound, and complete for $\mathbf{Calls}$. We have completely formalized these results (modulo well-known Horn clause logic definitions

and properties) in Coq [33], code for which can be provided by the authors upon request by the PC Chair. In this section we summarize our results.

Theorem 4.4.1 *Program rewriting algorithm $\mathcal{R}_{\Lambda_{\text{call}}}$ is semantics preserving (Definition 2.5.1).*

Proof. Intuitively, the addition of $callEvent(\mathbf{f}, v)$ and $emit(\mathbf{f}, v)$ commands does not interfere with Λ_{call} evaluation. The proof follows easily by induction on the number of small-step reductions of programs. □

Our proof strategy for soundness and completeness of $\mathcal{R}_{\Lambda_{\text{call}}}$ is to show that an audit log produced by an instrumented program is the refinement of the least Herbrand model of the logging specification semantics unioned with the logging specification's guidelines. By showing that audit logs combined with the guidelines are the least Herbrand models of the logging specification semantics, we show that they contain the same information. This implies soundness and completeness of program rewriting.

The following Lemma relates the syntactic property of closure with the properties of a least Herbrand model [34, 31], and shows that the least Herbrand model of X contains the same information as X. It holds by the soundness and completeness of the logic.

Lemma 4.4.1 $C(\mathfrak{H}(X)) = C(X)$ *and* $\mathfrak{H}(X) = \mathfrak{H}(C(X))$.

The following Lemmas states a similar but subtly different property relevant to sublanguage focusing that we will use in Theorem 4.4.2.

Lemma 4.4.2 $C(C(\mathfrak{H}(X)) \cap L) = C(\mathfrak{H}(X) \cap L)$.

The key idea underlying the soundness of the program rewriting algorithm is that any facts that are added to the set of logging preconditions or the audit log during execution of the instrumented program are true facts: they are in the model of the corresponding source language execution trace.

Lemma 4.4.3 *Let* $\mathfrak{p}$ *be a* Λ_{call} *program and* $LS \in \textbf{Calls}$ *be a logging specification. For all target language execution traces* τ *such that* $\mathcal{R}_{\Lambda_{\text{call}}}(\mathfrak{p}, LS) \Downarrow \tau$*, where* $\tau = \kappa_0 \ldots \kappa_n$ *and* $\kappa_n = (e, X, m, \mathbb{L}, \mathcal{C})$*, there exists a source language execution trace* τ' *such that* $\tau' :\approx \tau$ *and* $\mathfrak{p} \Downarrow \tau'$ *and* $X \subseteq toFOL(\tau')$*.*

To show that $\mathcal{R}_{\Lambda_{\text{call}}}$ is complete, we must show that for a logging specification $LS = spec(X_{Guidelines}, \{\text{LoggedCall}\}) \in \textbf{Calls}$ and a source language execution τ, and a corresponding target language execution τ' that produces audit log $\mathbb{L}$, for any ground instance $\text{LoggedCall}(t, \mathbf{f}, v) \in LS(\tau)$ we have $\text{LoggedCall}(t, \mathbf{f}, v) \in \mathbb{L}$. In order to show that, we need to show that $(X \cup X_{Guidelines}) \vdash \text{LoggedCall}(t, \mathbf{f}, v)$, where X is the set of logging preconditions tracked during the target language execution τ' (see Rules Precondition and Log).

A key insight is that the only facts in $toFOL(\tau)$ relevant to deriving grounded goals of the form $\text{LoggedCall}(t, \mathbf{f}, v)$ are facts $\text{Call}(t', \mathbf{f}', v')$ for $\mathbf{f}' \in \{Logevent(LS)\} \cup Triggers(LS)$, and these are exactly the facts that appear in the instrumented program's set of logging preconditions tracked during execution. Formally, the *support of* a grounded goal ψ given assumptions X, denoted $support(X, \psi)$, is the set of conjuncts in ϕ where $\phi \Rightarrow \psi$ is a grounding of a Horn clause $\forall x_1, \ldots, x_m.\ \phi' \Rightarrow \psi' \in X$ and $X \vdash \phi$. In Datalog terms, these are the grounded subgoals of ψ in its derivation given knowledge base X. Hence:

Lemma 4.4.4 *Let* $\mathfrak{p}$ *be a* Λ_{call} *program and* $LS \in \textbf{Calls}$ *be a logging specification where* $LS = spec(Y, S)$*. For all* τ *such that* $\mathfrak{p} \Downarrow \tau$ *there exists a target language execution trace* τ' *such that* $\tau :\approx \tau'$*,* $\mathcal{R}(\mathfrak{p}, LS) \Downarrow \tau'$ *and* $\tau' = \kappa_0 \ldots \kappa_n$ *where* $\kappa_n = (e, X, m, \mathbb{L}, \mathcal{C})$ *such that for all* $\phi \in LS(\tau)$ *and* $\text{Call}(t, \mathbf{g}, v) \in support(Y \cup toFOL(\tau), \phi)$ *we have* $\text{Call}(t, \mathbf{g}, v) \in X$*.*

From Lemma 4.4.3 and Lemma 4.4.4, we can establish that the log generated by the rewritten program is the least Herbrand model of the given logging specification semantics.

Lemma 4.4.5 *Let* $\mathfrak{p}$ *be a* Λ_{call} *program and* $LS = spec(X, \{\text{LoggedCall}\}) \in \textbf{Calls}$ *be a logging specification. For all* τ *such that* $\mathfrak{p} \Downarrow \tau$ *we have* $simlogs(\mathcal{R}_{\Lambda_{\text{call}}}(\mathfrak{p}, LS), \tau) = \{\mathbb{L}\}$ *such*

that:

$$\mathbb{L} = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{LoggedCall}\}}$$

*Proof. (*Sketch.) First, note that we can construct a target language execution trace τ' such that $\mathcal{R}_{\Lambda_{\text{call}}}(\mathfrak{p}, LS) \Downarrow \tau'$ and $\tau :\approx \tau'$ (i.e., τ' executes the source program to the same point that τ does). Let the last configuration of τ' be $(e, Y, n, \mathbb{L}, \mathcal{C})$. We observe that this construction uniquely defines the log $\mathbb{L}$ due to determinism in the language and Definition 4.3.1.

Let $Z = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{LoggedCall}\}}$. If $\text{LoggedCall}(n, \mathbf{f}, v) \in \mathbb{L}$, then $X \cup Y \vdash \text{LoggedCall}(n, \mathbf{f}, v)$ by semantic definition of $\mathcal{L}_{\text{log}}$. But by Lemma 4.4.3 we have $X \subseteq toFOL(\tau)$, hence $X \cup toFOL(\tau) \vdash \text{LoggedCall}(n, \mathbf{f}, v)$ so $\text{LoggedCall}(n, \mathbf{f}, v) \in Z$.

Conversely, if $\text{LoggedCall}(n, \mathbf{f}, v) \in Z$, by Lemma 4.4.4 (and the determinism of our languages), any Call fact in $support(X \cup toFOL(\tau), \text{LoggedCall}(n, \mathbf{f}, v))$ is also in X, hence every such LoggedCall will also be in $\mathbb{L}$. Thus $\text{LoggedCall}(n, \mathbf{f}, v) \in Z$ iff $\text{LoggedCall}(n, \mathbf{f}, v) \in \mathbb{L}$. The result follows by definition of $\mathfrak{H}$. □

These Lemmas suffice to prove our main Theorem, demonstrating soundness and completeness of program rewriting algorithm $\mathcal{R}_{\Lambda_{\text{call}}}$. This result establishes that the log generated by the instrumented program and the semantics of the logging specification contain exactly the same information with respect to the sublanguage $L_{\{\text{LoggedCall}\}}$.

Theorem 4.4.2 (Soundness and Completeness) *Program rewriting algorithm $\mathcal{R}_{\Lambda_{\text{call}}}$ is sound and complete (Definitions 2.5.2 and 2.5.3).*

Proof. Let $\mathfrak{p}$ be a Λ_{call} program and $LS = spec(X, \{\text{LoggedCall}\}) \in \mathbf{Calls}$ be a logging specification. We aim to show that for all source language execution traces τ such that $\mathfrak{p} \Downarrow \tau$ we have $simlogs(\mathcal{R}_{\Lambda_{\text{call}}}(\mathfrak{p}, LS), \tau) = \{\mathbb{L}\}$ such that $C(\mathbb{L}) = LS(\tau)$.

By Lemma 4.4.5, we have that $simlogs(\mathcal{R}_{\Lambda_{\text{call}}}(\mathfrak{p}, LS), \tau) = \{\mathbb{L}\}$ such that $\mathbb{L} = \mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\text{LoggedCall}\}}$. By Lemma 4.4.1 and Lemma 4.4.2 $LS(\tau) = C(C(\mathfrak{H}(X \cup$

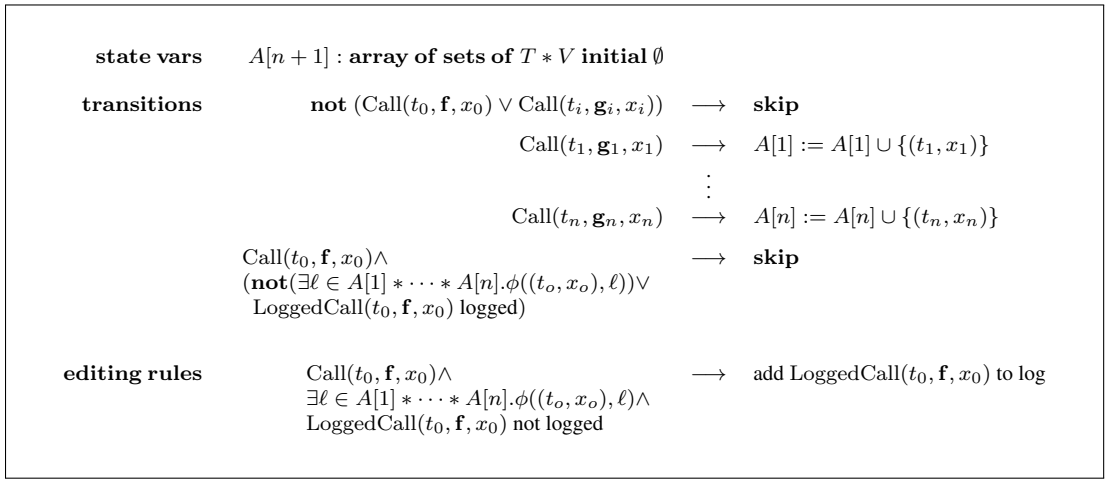

Figure 4.1: Edit automata that enforces ideal instrumentation

$toFOL(\tau))) \cap L_{\{\mathrm{LoggedCall}\}}) = C(\mathfrak{H}(X \cup toFOL(\tau)) \cap L_{\{\mathrm{LoggedCall}\}})$. Hence, both $LS(\tau) \leq C(\mathbb{L})$ and $C(\mathbb{L}) \leq LS(\tau)$. □

4.5 Edit Automata Enforcement of Calls Specifications

Following Theorem 2.4.1, we observe that, given a logging specification in **Calls**, we can easily define an edit automata that enforces this property. The following Definition is in the "guarded command" style used by Schneider [13]. The array A is used to store potentially multiple values of potentially multiple calls to each function g_i.

Theorem 4.5.1 *Given $spec(X, S) \in$ **Calls**, the ideal instrumentation property is enforced by the edit automata in Figure 4.1. It is defined using the following predicates on input configurations κ:*

$$\mathrm{Call}(t, f, x) \quad : \quad \textit{means } \mathrm{Call}(t, f, x) \in toFOL(\kappa)$$

$$\phi \textit{ logged} \quad : \quad \textit{means } \phi \in toFOL(logof(\kappa))$$

and also T and V denote the universes of timestamps and program values respectively.

Proof. Straightforward by induction on traces and definitions of edit automata [26]. □

However, as indicated in Section 2.4, this technique does not provide an adequate semantic foundation for log generation, and consequently correctness studies.

Chapter 5

Case Study on a Medical Records System

As a case study, we have developed a tool [35] that enables automatic instrumentation of logging specifications for the OpenMRS system. The implementation is based on the formal model developed in Chapter 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 [15] is a Java-based open-source web application for medical records, built on the Spring Framework [36]. 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 [24]. 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 [32]. This highlights the relevance of our $\mathbf{Calls}$ logging specification class, particularly as it pertains to specification of

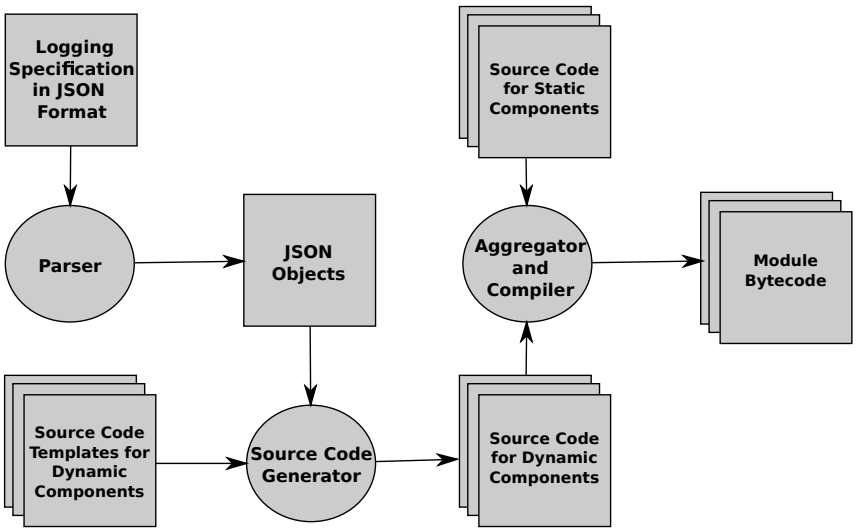

Figure 5.1: Module builder

break the glass policies, which are sensitive to authorization.

In contrast to the earlier auditing solutions for OpenMRS, ours facilitates a smart log generation mechanism in which only the necessary information are recorded, based on accurate log specifications. Moreover, logging specifications are defined independently from code, rather than being embedded in it in an ad-hoc manner. This way, system administrators need to only assert logging specifications in the style of **Calls** (Definition 4.2.1), and the tool builds the corresponding module that could be installed on the OpenMRS server. This is more convenient, declarative, and less error prone than direct ad-hoc instrumentation of code. In Figure 5.1 the details of building the module is given.

System Architecture Summary To clarify the following discussion, we briefly summarize the architecture of our system. Logging specifications are made in the style of **Calls**, 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

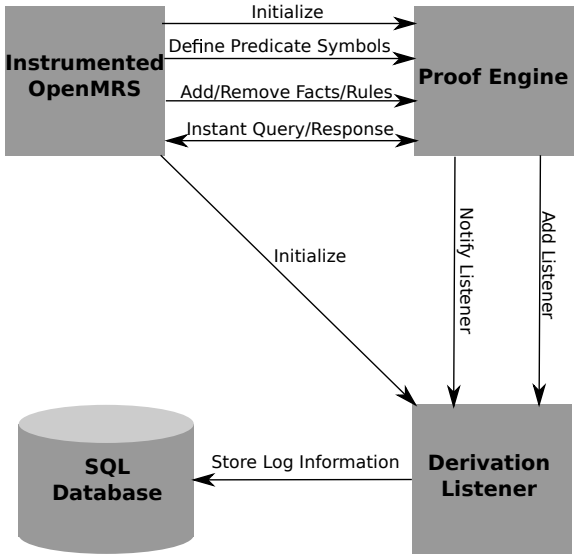

Figure 5.2: System architecture

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. Figure 5.2 illustrates the major components we have deployed to facilitate auditing at runtime.

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 “Encounter” domain

includes the 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 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 [32]. Likewise, we focus on these functionalities to be addressed in the logging specifications, i.e., the triggers and logging events are constrained to the service-layer methods as they provide access to data domains, e.g., the patient and user data.

We adapt the logical language of logging specifications developed above (Definition 4.2.1), 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, and generally define predicates specified in $\phi((x_0, t_0), \ldots, (x_n, t_n))$ of Definition 4.2.1. We consider break the glass policies as a key example application in our deployment. For instance a simple break the glass policy states that if the glass is broken by some low-level user, and subsequently the 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 work.

It is worth mentioning that while our tool is designed for OpenMRS, our general approach can be used for arbitrary Java code at source or bytecode level.

5.1.1 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.

Since the generated code pieces are before advices, they are interposed before every interface method of the declared services. An aspect is configured by declaring where the join point and corresponding advice is. 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.

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

The advice `RetroSecurityAdvice` is the before advice automatically generated by our system based on the logging specification. It essentially determines whether a method call is a trigger or a logging event and interacts with the proof engine appropriately in each case.

The first time the advice is executed, the XSB Prolog engine is initialized in a separate thread. Moreover, a $\mathrm{LoggedCall}$ derivation listener is added to the list of the engine listeners. Then, if memory overhead mitigation (Section 5.2) is not activated, the invoked method names are checked and the rule $\mathrm{Protection}$ (Section 4.3) is implemented for the triggers and the logging event, i.e., the proof engine is asked to add the the information regarding the invocation of the method. In the case memory overhead mitigation is activated, the set of $\mathrm{Protection}$ rules of Figure 5.3 are implemented for the triggers and the logging event. The implementation of the rules Log and NoLog (Section 4.3) is handled by the $\mathrm{LoggedCall}$ derivation listener.

The advice also checks for the invocation of the interface method `queryLog()`. This method communicates with the engine to facilitate instant querying based on the invocations of the logging preconditions that exist in the memory.

5.1.2 Proof Engine

According to the the semantics of $\Lambda_{\log}$, it is necessary to perform logical deduction, in particular resolution of $\mathrm{LoggedCall}$ predicates. As we will show in Section 5.2, the required deductions could be generalized to any arbitrary formula. To this end, we have employed XSB Prolog [37] as our proof engine, due to its reliability and robustness. We have restricted our specifications to safe Horn clauses though, despite the fact that XSB Prolog provides a more expressive tool. In order to have a bidirectional communication between the Java application and the engine, InterProlog Java/Prolog SDK [38] 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, to add rules and facts, and to revoke

them asynchronously[1]. The asynchrony avoids blocking the "normal" execution trace for audit logging purposes. The interface also provides an instant querying mechanism. The execution trace of instrumented program communicates with the XSB Prolog engine as these interface methods are invoked in the advices.

5.1.3 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. The generated advices include event listeners to implement our technology of choice for log storage and retainment. In case a 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, i.e., the join of these tables are information equivalent (Definition 2.1.4) to the semantics of logging specification for a given break the glass policy. This ensures that the correctness guarantees extend to database storage.

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

[1]Revoking facts is required for memory overhead mitigation.

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
```

5.2 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, we can limit the amount of information required to derive that conjunct to the point where it is derivable, without affect-

ing the derivability of other conjuncts. In other words, following derivation of that conjunct, triggers in the conjunct are "turned off", i.e. no longer added to logging preconditions when encountered during execution.

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.

Since we have a linear computational model, the predicates corresponding to the timestamp comparisons ($t_i < t_0$) do not play a significant role in the inference of $\mathrm{LoggedCall}$ predicates. There reason is, at any point in time, the set of logging preconditions only contain function invocations that have been occurred in the past, i.e., if the logging event is invoked at timestamp t_0, then $t_i < t_0$ holds for all trigger invocation timetamps t_i that are retained in the set of logging preconditions. In what follows, a formal description of the technique is given.

Consider the Definition 4.2.1, where $\mathbf{g}_0 = Logevent(LS)$. Let $\varphi_{i'}((x_0, t_0), \ldots, (x_n, t_n))$s be positive literals and $\phi((x_0, t_0), \ldots, (x_n, t_n)) \triangleq \bigwedge_{i'=1}^{n'} \varphi_{i'}((x_0, t_0), \ldots, (x_n, t_n))$. Then, we define the set $\Psi \triangleq \{\varphi_{i'}((x_0, t_0), \ldots, (x_n, t_n)) \mid i' \in 1 \cdots n'\} \cup \{(\mathrm{Call}(t_i, \mathbf{g}_i, x_i)) \mid i \in 0 \cdots n\}$. 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. Then, $FV(\Psi) = \{x_0, \cdots, x_n, t_0, \cdots, t_n\}$. 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.

Definition 5.2.1 *Let $\circledast_{FV} \subseteq FV(\Psi) \times FV(\Psi)$ be a relation where $\alpha \circledast_{FV} \beta$ iff there exists some literal $P \in \Psi$ such that $\alpha, \beta \in FV(P)$. Then, the transitive closure of $\circledast_{FV}$ is denoted by $\circledast_{TFV}$.*

Lemma 5.2.1 *$\circledast_{FV}$ is reflexive and symmetric.*

Corollary 5.2.1 *$\circledast_{TFV}$ is an equivalence relation and so, partitions $FV(\Psi)$*

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 5.2.2). 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 $\omega : FV(\Psi) \rightarrow \{1, \cdots, L\}$ where $\omega(\alpha) = k$ if $[\alpha]_{\circledast_{TFV}} = C_k$.

Definition 5.2.2 *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 $\mathcal{P}_C \triangleq \{P \in \Psi \mid FV(P) \subseteq C\}$. We define the independent conjuncts as $Q_C \triangleq \bigwedge_{P \in \mathcal{P}_C} P$. We also denote $Q_{[\alpha]}$ as Q_k if $\omega(\alpha) = k$. Obviously, $FV(Q_k) = C_k$.*

Lemma 5.2.2 *Let $C_1, \cdots, C_L$ be all the equivalence classes induced by $\circledast_{TFV}$ over $FV(\Psi)$. Then, $\mathcal{P}_{C_1}, \cdots, \mathcal{P}_{C_L}$ give a partition on Ψ.*

Proof. We need to show that

- for all distinct $k, k' \in 1 \cdots L$, $\mathcal{P}_{C_k} \cap \mathcal{P}_{C_{k'}} = \emptyset$:

  By contradiction: Let $k, k' \in 1 \cdots L$ be specific distinct indexes where $\mathcal{P}_{C_k} \cap \mathcal{P}_{C_{k'}} \neq \emptyset$, i.e., there exists some $P \in \Psi$, such that $P \in \mathcal{P}_{C_k}$ and $P \in \mathcal{P}_{C_{k'}}$. Then, according to the definition, we have $FV(P) \subseteq C_k$ and $FV(P) \subseteq C_{k'}$. Since $FV(P)$ is non-empty, we would have $C_k \cap C_{k'} \neq \emptyset$, which contradicts with C_k and $C_{k'}$ being classes over $FV(\Psi)$.

- $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} = \Psi$:

  Obviously, $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} \subseteq \Psi$by the definition of predicate classes. It only suffices to show that $\bigcup_{k=1}^{L} \mathcal{P}_{C_k} \supseteq \Psi$. Let $P \in \Psi$. Since $FV(P) \neq \emptyset$, there exists some $\alpha \in FV(P)$. Considering the equivalence class $[\alpha]_{\circledast_{TFV}}$, we will then have $FV(P) \subseteq [\alpha]_{\circledast_{TFV}}$. This entails that $P \in \mathcal{P}_{[\alpha]_{\circledast_{TFV}}}$ and so, $P \in \bigcup_{k=1}^{L} \mathcal{P}_{C_k}$.

□

In order to specify and prove the correctness of the proposed technique, a new calculus $\Lambda'_{\log}$ is formalized with memory overhead mitigation capabilities. In what follows the details of this calculus and the correctness result are given. Moreover, a developed example of how these techniques could be applied to a sample logging specification in **Calls** is discussed, later in this Section.

The given techniques are implemented in the OpenMRS retrospective security module as a case study.

5.2.1 Language with Memory Overhead Mitigation

The language $\Lambda_{\log}$ is given in Chapter 4 whose syntax includes a command to track logging preconditions ($callEvent(\mathbf{g}_i, v)$) and a command to emit log entries ($emit(\mathbf{g}_i, v)$). Configurations are quintuples of the form $\kappa ::= (e, X, n, \mathbb{L}, \mathcal{C})$ which include a set X of logging precondtions (sometimes referred to as "database"), and an audit log $\mathbb{L}$. The semantics of $\Lambda_{\log}$ includes the rule Precondition to update the set of logging preconditions.

The language $\Lambda'_{\log}$ has the same syntax as $\Lambda_{\log}$. The configurations, however, have an additional component W which is a set of function names. It is used to keep track of functions that we do not require to add their invocation information to the database any more. By adding some trigger name $\mathbf{g}_i$ to W, we indicate that further additions of information regarding $\mathbf{g}_i$

invocations to X will not cause new $\mathrm{LoggedCall}$ predicates to be derived.

$$\kappa ::= (e, X, n, \mathbb{L}, \mathcal{C}, W) \qquad \textit{configurations}$$

All stepwise reduction rules in this language are the same as the ones in $\Lambda_{\log}$, except for Precondition. Instead of that rule, we impose the set of rules in Figure 5.3. Note that X_G denotes the guidelines database. For the sake of brevity, we will omit $\circledast_{TFV}$ from the class notations onward.

The rule Precondition-1 states that if a trigger is invoked, but is already added to the set W, according to the semantics of W, we do not add the invocation to the database. The remaining rules consider the other case, i.e., the trigger is not already added to W. The rule Precondition-2 expresses the case where the trigger $\mathbf{g}_i$ is not in W, and there are no literals except for $\mathrm{Call}(t_i, \mathbf{g}_i, x_i)$ with x_i or t_i as free variables. In this case, the invocation is added to database and the trigger name is added to the set W, in order to avoid further addition of invocations to this trigger. If there are literals other than $\mathrm{Call}(t_i, \mathbf{g}_i, x_i)$ with free variables x_i or t_i, but the free variables of all those literals are restricted to x_i and t_i, we study the derivability of the ground form of $Q_{[t_i]}$ considering the new invocation. Notice that $FV(Q_{[t_i]}) = \{x_i, t_i\}$. If the ground form of $Q_{[t_i]}$ is derivable, then the invocation is added to the database. The trigger name is also added to W (Precondition-4). Otherwise, the invocation is not added to the database (Precondition-3). The reason is, keeping the invocation information in the database will not help deriving a ground form of $Q_{[t_i]}$ in the future steps.

If there are literals other than $\mathrm{Call}(t_i, \mathbf{g}_i, x_i)$ with free variables x_i or t_i, and the free variables of those literals are not restricted to x_i and t_i, but exclude x_0 and t_0, then the derivability of $Q_{[t_i]}$ is studied. In this case, the set of free variables of $Q_{[t_i]}$ is $[t_i]$, for sure. If a ground form of $Q_{[t_i]}$ is derivable, then the invocation is added to the database and the trigger names whose timestamp and argument variable are in $[t_i]$ are added to W (Precondition-6). Otherwise, the

Precondition-1

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \in W}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X, n, \mathbb{L}, \mathcal{C}, W)}$$

Precondition-2

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} = \emptyset}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C}, W \cup \{\mathbf{g}_i\})}$$

Precondition-3

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W}{\mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} \neq \emptyset \qquad [t_i] - \{x_i, t_i\} = \emptyset \qquad X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\} \cup X_G \nvdash Q_{[t_i]}\big[n-1/t_i\big]\big[v/x_i\big]}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X, n, \mathbb{L}, \mathcal{C}, W)}$$

Precondition-4

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W}{\mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} \neq \emptyset \qquad [t_i] - \{x_i, t_i\} = \emptyset \qquad X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\} \cup X_G \vdash Q_{[t_i]}\big[n-1/t_i\big]\big[v/x_i\big]}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C}, W \cup \{\mathbf{g}_i\})}$$

Precondition-5

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} \neq \emptyset}{[t_i] - \{x_i, t_i\} \neq \emptyset \qquad [t_i] - \{x_0, t_0\} = \emptyset \qquad \nexists \bar{a}_i \,.\, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\} \cup X_G \vdash Q_{[t_i]}\big[\bar{a}_i/\bar{\alpha}_i\big]\big[n-1/t_i\big]\big[v/x_i\big]}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C}, W)}$$

Precondition-6

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} \neq \emptyset}{[t_i] - \{x_i, t_i\} \neq \emptyset \qquad [t_i] - \{x_0, t_0\} = \emptyset \qquad \exists \bar{a}_i \,.\, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\} \cup X_G \vdash Q_{[t_i]}\big[\bar{a}_i/\bar{\alpha}_i\big]\big[n-1/t_i\big]\big[v/x_i\big]}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C}, W \cup_{t_{i'} \in [t_i]} \{\mathbf{g}_{i'}\})}$$

Precondition-7

$$\frac{i \in 1 \cdots n \qquad \mathbf{g}_i \notin W \qquad \mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} \neq \emptyset \qquad [t_i] - \{x_i, t_i\} \neq \emptyset \qquad [t_i] - \{x_0, t_0\} \neq \emptyset}{(\mathit{callEvent}(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C}, W)}$$

Precondition-8

$$\frac{}{(\mathit{callEvent}(\mathbf{g}_0, v); e, X, n, \mathbb{L}, \mathcal{C}, W) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_0, v)\}, n, \mathbb{L}, \mathcal{C}, W)}$$

Figure 5.3: Precondition rules for $\Lambda'_{\log}$.

invocation is still added to the database (Precondition-5) but the trigger name is not added to W. The reason is, keeping the invocation information in the database might help derive a ground form of $Q_{[t_i]}$ in the future steps, since there exist free variables other than x_i and t_i in $Q_{[t_i]}$ that could be substituted with proper values so that $Q_{[t_i]}$ could be derived. Note that $\bar{\alpha}_i$ represents a sequence of free variables $[t_i] - \{x_i, t_i\}$ and $\bar{a}_i$ is a sequence of timestamps and values, except for the timestamp and argument value of trigger i. Moreover, $\left[\bar{a}_i/\bar{\alpha}_i\right]$ denotes the substitution of values to their corresponding variables.

The rule Precondition-7 discusses the remaining case for triggers, that is when there are literals other than $\mathrm{Call}(t_i, \mathbf{g}_i, x_i)$ with free variables x_i or t_i, the free variables of those literals are not restricted to x_i and t_i, and include x_0 or t_0. Then, the invocation is added to the database but the trigger name is not added to W, independent of whether a ground form of $Q_{[t_i]}$ is derivable or not. If a ground form of $Q_{[t_i]}$ is not derivable at the moment, keeping the invocation information in the database might help derive a ground form of $Q_{[t_i]}$ in the future steps, since there exist free variables other than x_i and t_i in $Q_{[t_i]}$ that could be substituted with proper values so that a ground form of $Q_{[t_i]}$ could be derived. Otherwise, if a ground form of $Q_{[t_i]}$ is derivable, we might still need to add future invocations of $\mathbf{g}_i$ and other triggers whose timestamp and argument variables are in $[t_i]$. That is why, we avoid adding trigger names to W. This is due to the fact that $Q_{[t_i]}$ includes invocation to the logging event and possibly other predicates defined over its timestamp and argument variable (t_0 and x_0). Thus, future derivations of $Q_{[t_i]}$ could be affected.

This represents a major difference between the case when $Q_{[t_i]}$ includes $\{x_0, t_0\}$ and the case $Q_{[t_i]}$ excludes these variables. In the latter case, it is only required to derive a ground form of $Q_{[t_i]}$ once during program execution, in order to study whether $\mathrm{LoggedCall}$ predicates could be inferred. Therefore, whenever a ground form of $Q_{[t_i]}$ is derivable at the time of $\mathbf{g}_i$ invocation, W is beefed up with the corresponding trigger names. In the prior case, however,

it is required to derive all possible ground forms of $Q_{[t_i]}$, so that we would be able to infer all possible $\mathrm{LoggedCall}$ predicates.

The last rule (Precondition-8) discusses the case where the logging event is invoked. Since we need to infer all possible $\mathrm{LoggedCall}$ predicates, we add all those invocations to the database.

5.2.2 Correctness of Memory Overhead Mitigation

In order to study the executional behaviour of programs in $\Lambda'_{\log}$ compared to the case where they are executing in $\Lambda_{\log}$, we need to understand the relationship between the set of logging preconditions in these languages. To this end, we develop an algorithm that generates the reduced database and the set W of trigger names, out of a full-blown database of logging preconditions. The algorithm Refine is defined in Figure 5.4. We denote the reduced set of logging preconditions X as $\mathcal{R}(X)$, and the generated set of trigger names as $\mathcal{W}(X)$, defined as follows.

$$\mathcal{R}(X) \triangleq \mathrm{fst}(\mathrm{Refine}\ X\ [\,]\ \emptyset),$$
$$\mathcal{W}(X) \triangleq \mathrm{snd}(\mathrm{Refine}\ X\ [\,]\ \emptyset).$$

We do not express any explicit mapping between sets and sorted lists in our formulation for the sake of brevity. The employment of sets and their corresponding sorted lists are clear from the context. Let's denote the restriction of a set X to timestamps less than or equal to n, as $X|_n$.

In what follows, Lemmas 5.2.3 to 5.2.7 discuss properties of $\mathcal{R}(X)$ and $\mathcal{W}(X)$. Lemma 5.2.7, in particular, shows that $\mathcal{R}(X)$ is enough to derive all $\mathrm{LoggedCall}$ predicates that are derivable from X. Lemma 5.2.8 states that in a single reduction step, the reduced set of logging preconditions is preserved and the generated audit log is maintained, which then can be

```
1: function REFINE: Sorted list of invocation facts → Sorted list of invocation facts → Set of trigger names → (Sorted list of
   invocation facts × Set of trigger names)
2: Refine [ ] Y W = (Y, W)
3: Refine ((Call(n, g_i, v)) :: X) Y W =
4:     if i ∈ 1 ··· n then
5:         if g_i ∈ W then
6:             Refine X Y W
7:         else
8:             if P_[t_i] − {Call(t_i, g_i, x_i)} = ∅ then
9:                 Refine X (Y + [Call(n, g_i, v)]) (W ∪ {g_i})
10:            else
11:                if [t_i] − {x_i, t_i} = ∅ then
12:                    if Y ∪ {Call(n − 1, g_i, v)} ∪ X_G ⊬ Q_[t_i][n − 1/t_i][v/x_i] then
13:                        Refine X Y W
14:                    else
15:                        Refine X (Y + [Call(n, g_i, v)]) (W ∪ {g_i})
16:                    end if
17:                else
18:                    if [t_i] ∩ {x_0, t_0} = ∅ then
19:                        if ∄ā_i . Y ∪ {Call(n, g_i, v)} ∪ X_G ⊢
                                  Q_[t_i][ā_i/ᾱ_i][n/t_i][v/x_i] then
20:                            Refine X (Y + [Call(n, g_i, v)]) W
21:                        else
22:                            Refine X (Y + [Call(n, g_i, v)]) (W ∪_{t_i' ∈ [t_i]} {g_i'})
23:                        end if
24:                    else
25:                        Refine X (Y + [Call(n, g_i, v)]) W
26:                    end if
27:                end if
28:            end if
29:        end if
30:    else
31:        Refine X (Y + [Call(n, g_i, v)]) W
32:    end if
33: end function
```

Figure 5.4: Refine algorithm

generalized straightforwardly to multi-step reduction, in Theorem 5.2.1. Then, Corollary 5.2.2 gives us our intended result, which states that a program could be evaluated in $\Lambda'_{\log}$ with reduced set of logging preconditions and the same audit log.

Lemma 5.2.3 *Let X be a set of logging preconditions. For all $i \in 1 \cdots n$, if $t_i \in [t_0]$ then $\mathbf{g}_i \notin \mathcal{W}(X)$.*

Proof. Since $t_i \in [t_0]$, $[t_i] = [t_0]$ and so $[t_i] - \{x_0, t_0\} \neq \emptyset$. Thus, for each invocation information of $\mathbf{g}_i$ in X, only line 25 of Figure 5.4 is executed. Obviously, $\mathbf{g}_i$ is not added to W in this line. □

Lemma 5.2.4 *For all $i \in 0 \cdots n$, if $t_i \in [t_0]$ and $\mathrm{Call}(n, \mathbf{g}_i, v) \in X$, then $\mathrm{Call}(n, \mathbf{g}_i, v) \in \mathcal{R}(X)$.*

Proof. First let's consider the case where $i = 0$. Then, according to line 31 of Figure 5.4, $\mathrm{Call}(n, \mathbf{g}_i, v) \in \mathcal{R}(X)$. Now, let $i \in 1 \cdots n$. According to Lemma 5.2.3, $\mathbf{g}_i \notin \mathcal{W}(X)$ and so $\mathbf{g}_i \notin \mathcal{W}(X|_{n-1})$. This implies that only line 25 of Figure 5.4 is can be executed for $\mathbf{g}_i$, in which the invocation to $\mathbf{g}_i$ is added to Y, which then is reflected in $\mathcal{R}(X)$. □

Let $\bar{t}(k)$ and $\bar{x}(k)$ denote sequences of timestamp and function argument variables respectively, that are in class C_k. Similarly, $\bar{s}(k)$ and $\bar{v}(k)$ are used to denote sequences of timestamp and function argument values that substitute $\bar{t}(k)$ and $\bar{x}(k)$.

Lemma 5.2.5 *Let $i \in 1 \cdots n$. Suppose $X \cup X_G \vdash Q_{\omega(i)}\left[\bar{s}(\omega(i))/\bar{t}(\omega(i))\right]\left[\bar{v}(\omega(i))/\bar{x}(\omega(i))\right]$ for some timestamp and argument value sequences $\bar{s}(\omega(i))$ and $\bar{v}(\omega(i))$. If $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \in X$ and $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \notin \mathcal{R}(X)$ for some i' such that $t_{i'} \in [t_i]$, then $\mathbf{g}_{i'} \in \mathcal{W}(X|_{s_{i'}-1})$.*

Proof. Let's assume $\mathbf{g}_{i'} \notin \mathcal{W}(X|_{s_{i'}-1})$. Since $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \in X$ and $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \notin \mathcal{R}(X)$, we need to follow Refine algorithm to extract the places where the invocation information is not added to Y. The only place with such a property is line 13 (other than line 6 which

is refuted by the assumption). Then, $\mathcal{P}_{[t_{i'}]} - \{\mathrm{Call}(t_{i'}, \mathbf{g}_{i'}, x_{i'})\} = \emptyset$, $[t_{i'}] - \{x_{i'}, t_{i'}\} = \emptyset$, and $X|_{s_{i'}-1} \cup \{\mathrm{Call}(s_{i'} - 1, \mathbf{g}_{i'}, v_{i'})\} \cup X_G \nvdash Q_{[t_{i'}]}\big[s_{i'} - 1/t_{i'}\big]\big[v_{i'}/x_{i'}\big]$. The latter result is in contradiction with the general form $X \cup X_G \vdash Q_{\omega(i)}\big[\bar{s}(\omega(i))/\bar{t}(\omega(i))\big]\big[\bar{v}(\omega(i))/\bar{x}(\omega(i))\big]$ considering the fact that $\omega(i) = \omega(i')$ due to $t_{i'} \in [t_i]$, and also $Q_{[t_i]} = Q_{\omega(i')}$. □

Lemma 5.2.6 *Let X be a set of logging preconditions. For all $i \in 1 \cdots n$, if $\mathbf{g}_i \in \mathcal{W}(X)$, then there exist some n, v, and $\bar{a}_i$ such that $\mathrm{Call}(n, \mathbf{g}_i, v) \in \mathcal{R}(X)$ and*

$$\mathcal{R}(X) \cup X_G \vdash Q_{[t_i]}\big[\bar{a}_i/\bar{\alpha}_i\big]\big[n/t_i\big]\big[v/x_i\big].$$

Proof. The only places where a trigger is added to W in Figure 5.4 are the lines 9, 15, and 22. In line 9, invocation to $\mathbf{g}_i$ is added to Y, which then is reflected in $\mathcal{R}(X)$. Moreover, this line is executed whenever $\mathcal{P}_{[t_i]} - \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i)\} = \emptyset$. Thus, for this case $Q_{[t_i]} = \mathrm{Call}(t_i, \mathbf{g}_i, x_i)$. Then, since $\mathrm{Call}(n, \mathbf{g}_i, v) \in \mathcal{R}(X)$, $\mathcal{R}(X) \cup X_G \vdash \mathrm{Call}(n, \mathbf{g}_i, v)$. In line 15, similar to line 9, invocation to $\mathbf{g}_i$ is added to Y, so $\mathrm{Call}(n, \mathbf{g}_i, v) \in \mathcal{R}(X)$. Moreover, this line is executed provided the condition in lines 12 does not hold. This ensures the derivation of the ground form of $Q_{[t_i]}$. The line 22 is executed if the condition in lines 19 does not hold. Therefore, a ground form of $Q_{[t_i]}$ should be derivable. This entails that for all $t_{i'} \in [t_i]$, there exist some n' and v' such that $\mathrm{Call}(n', \mathbf{g}_{i'}, v') \in Y \cup \{\mathrm{Call}(n, \mathbf{g}_i, v)\}$. As $Y \cup \{\mathrm{Call}(n, \mathbf{g}_i, v)\}$ is reflected in $\mathcal{R}(X)$, the proof is complete. □

Lemma 5.2.7 *If $X \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathbf{g}_0, v_0)$ then $\mathcal{R}(X) \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathbf{g}_0, v_0)$.*

Proof. $X \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathbf{g}_0, v_0)$ entails that there exist $s_1, \cdots, s_n, v_1, \cdots, v_n$ such that $X \cup X_G \vdash \bigwedge_{i=1}^{n}(s_i < s_0) \bigwedge_{k=1}^{L} Q_k\big[\bar{s}(k)/\bar{t}(k)\big]\big[\bar{v}(k)/\bar{x}(k)\big]$. It implies that for all $i \in 0 \cdots n$, $\mathrm{Call}(s_i, \mathbf{g}_i, v_i) \in X$. For each $i \in 0 \cdots n$, we consider the following two cases:

- $[t_i] = [t_0]$: Then, $t_i \in [t_0]$. Since $\mathrm{Call}(s_i, \mathbf{g}_i, v_i) \in X$, Lemma 5.2.4 implies that $\mathrm{Call}(s_i, \mathbf{g}_i, v_i) \in \mathcal{R}(X)$. Then,

$$\mathcal{R}(X) \cup X_G \vdash (s_i < s_0) \wedge Q_{\omega(i)}\big[\bar{s}(\omega(i))/\bar{t}(\omega(i))\big]\big[\bar{v}(\omega(i))/\bar{x}(\omega(i))\big].$$

- $[t_i] \neq [t_0]$: Then, $[t_i] - \{x_0, t_0\} = \emptyset$. Now, we consider the following two subcases:

  - For all $t_{i'} \in [t_i]$, if $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \in X$ then $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \in \mathcal{R}(X)$. Then obviously $(s_i < s_0)$ and $Q_{\omega(i)}\big[\bar{s}(\omega(i))/\bar{t}(\omega(i))\big]\big[\bar{v}(\omega(i))/\bar{x}(\omega(i))\big]$ are provable under assumption $\mathcal{R}(X) \cup X_G$.
  - There exists some $t_{i'} \in [t_i]$ such that $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \in X$ and $\mathrm{Call}(s_{i'}, \mathbf{g}_{i'}, v_{i'}) \notin \mathcal{R}(X)$. Then, by Lemma 5.2.5, $\mathbf{g}_{i'} \in \mathcal{W}(X|_{s_{i'}-1})$, which implies that $\mathbf{g}_{i'} \in \mathcal{W}(X)$. Using Lemma 5.2.6, we conclude that there exist some $n \leq s_{i'} - 1$, v, and $\bar{a}_{i'}$ such that $\mathrm{Call}(n, \mathbf{g}_{i'}, v) \in \mathcal{R}(X)$ and $\mathcal{R}(X) \cup X_G \vdash Q_{[t_{i'}]}\big[\bar{a}_{i'}/\bar{\alpha}_{i'}\big] \bigcap n/t_{i'}\big]\big[v/x_{i'}\big]$. This entails that $\mathcal{R}(X) \cup X_G \vdash (n < s_0) \wedge Q_{\omega(i)}\big[\bar{a}_{i'}/\bar{\alpha}_{i'}\big]\big[n/t_{i'}\big]\big[v/x_{i'}\big]$ considering the fact that $Q_{\omega(i)} = Q_{[t_{i'}]}$, as $t_{i'} \in [t_i]$.

Thus, $\mathcal{R}(X) \cup X_G$ suffices to derive ground forms of all $Q_{\omega(i)}$ and therefore $\mathcal{R}(X) \cup X_G \vdash \mathrm{LoggedCall}(s_0, \mathbf{g}_0, v_0)$. □

Lemma 5.2.8 *If* $(e, X, n, \mathbb{L}, \mathcal{C}) \to (e', X', n', \mathbb{L}', \mathcal{C})$ *in* $\Lambda_{\log}$*, then* $(e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \to (e', \mathcal{R}(X'), n', \mathbb{L}', \mathcal{C}, \mathcal{W}(X'))$ *in* $\Lambda'_{\log}$.

Proof. By induction on the derivation of $(e, X, n, \mathbb{L}, \mathcal{C}) \to (e', X', n', \mathbb{L}', \mathcal{C})$. The interesting cases are the reduction of $callEvent(\mathbf{g}_i, v)$ and $emit(\mathbf{g}_i, v)$.

Let $(callEvent(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}) \to (e, X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\}, n, \mathbb{L}, \mathcal{C})$. There are eight cases in $\Lambda'_{\log}$, then. The first case is where $i \in 1 \cdots n$ and $\mathbf{g}_i \in \mathcal{W}(X)$. Then, $(callEvent(\mathbf{g}_i, v); e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \to (e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X))$. Then we need to

show that $\mathcal{R}(X) = \mathcal{R}(X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\})$ and $\mathcal{W}(X) = \mathcal{W}(X \cup \{\mathrm{Call}(n-1, \mathbf{g}_i, v)\})$ for this case, which hold based on line 6 of Refine in Figure 5.4. The other seven cases are similarly provable based on the definition of Refine.

Let $(emit(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}) \rightarrow (e, X, n, \mathbb{L} \cup \{\mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)\}, \mathcal{C})$. This holds when $X \cup X_G \vdash \mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)$. Using Lemma 5.2.7, we then have $\mathcal{R}X \cup X_G \vdash \mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)$, which implies that $(emit(\mathbf{g}_i, v); e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \rightarrow (e, \mathcal{R}(X), n, \mathbb{L} \cup \{\mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)\}, \mathcal{C}, \mathcal{W}(X))$.

Let $(emit(\mathbf{g}_i, v); e, X, n, \mathbb{L}, \mathcal{C}) \rightarrow (e, X, n, \mathbb{L}, \mathcal{C})$. Then $X \cup X_G \nvdash \mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)$. Since $\mathcal{R}(X) \subseteq X$ and the proof system is monotone, we conclude that $\mathcal{R}(X) \cup X_G \nvdash \mathrm{LoggedCall}(n-1, \mathbf{g}_i, v)$. It then implies that $(emit(\mathbf{g}_i, v); e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \rightarrow (e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X))$.

□

Theorem 5.2.1 *If* $(e, X, n, \mathbb{L}, \mathcal{C}) \rightarrow^* (e', X', n', \mathbb{L}', \mathcal{C})$ *in* $\Lambda_{\log}$*, then*

$$(e, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X)) \rightarrow^* (e', \mathcal{R}(X'), n', \mathbb{L}', \mathcal{C}, \mathcal{W}(X'))$$

in $\Lambda'_{\log}$.

Proof. It is straightforward by induction on the derivation of $(e, X, n, \mathbb{L}, \mathcal{C}) \rightarrow^* (e', X', n', \mathbb{L}', \mathcal{C})$ using the result of Lemma 5.2.8. □

The following corollary states the correctness in the sense that a program could be evaluated in $\Lambda'_{\log}$ with reduced set of logging preconditions and the same audit log as it is evaluated in $\Lambda_{\log}$.

Corollary 5.2.2 *If* $(e, \emptyset, 0, \emptyset, \mathcal{C}) \rightarrow^* (v, X, n, \mathbb{L}, \mathcal{C})$ *in* $\Lambda_{\log}$*, then*

$$(e, \emptyset, 0, \emptyset, \mathcal{C}, \emptyset) \rightarrow^* (v, \mathcal{R}(X), n, \mathbb{L}, \mathcal{C}, \mathcal{W}(X))$$

in $\Lambda'_{\log}$.

5.2.3 An Example

Consider the following logging specification.

$$\forall t_0, \ldots, t_4, x_0, \ldots, x_4 \,.\, \mathrm{Call}(t_0, \mathbf{g}_0, x_0) \bigwedge_{i=1}^{4} (\mathrm{Call}(t_i, \mathbf{g}_i, x_i) \wedge t_i < t_0) \wedge$$
$$t_1 < t_2 \wedge x_1 \% 2 = 0 \wedge x_3 = x_4 \wedge \mathrm{HasSecLevel}(x_4, \mathbf{secret}) \wedge$$
$$\mathrm{Includes}(x_0, x_4) \implies \mathrm{LoggedCall}(t_0, \mathbf{g}_0, x_0).$$

Then,

$$\Psi = \{\mathrm{Call}(t_i, \mathbf{g}_i, x_i) \mid i \in 0 \cdots 4\} \cup \{t_1 < t_2, x_1 \% 2 = 0, x_3 = x_4,$$
$$\mathrm{HasSecLevel}(x_4, \mathbf{secret}), \mathrm{Includes}(x_0, x_4)\}$$

and obviously $FV(\Psi) = \{t_i, x_i \mid i \in 0 \cdots 4\}$. Note that each literal could be defined intensionally or extensionally beside the guideline.

Then, the relation $\circledast_{FV}$ includes several pairs including $(t_1, t_2), (x_3, x_4), (x_4, x_0)$ and (t_i, x_i) for all $i \in \{0, \cdots, 4\}$

We then have the following equivalence classes:

$$C_1 = \{t_1, x_1, t_2, x_2\},$$
$$C_2 = \{t_0, x_0, t_3, x_3, t_4, x_4\}.$$

Note that $\omega(1) = \omega(2) = 1$ and $\omega(0) = \omega(3) = \omega(4) = 2$. As an example, $\bar{t}(2) = t_0, t_3, t_4$ is a possible sequence of timestamp variables of class C_2.

This implies the following predicate classes in Ψ:

$$P_{C_1} = \{\mathrm{Call}(t_1, \mathbf{g}_1, x_1), \mathrm{Call}(t_2, \mathbf{g}_2, x_2), t_1 < t_2, x_1 \% 2 = 0\},$$
$$P_{C_2} = \{\mathrm{Call}(t_0, \mathbf{g}_0, x_0), \mathrm{Call}(t_3, \mathbf{g}_3, x_3), \mathrm{Call}(t_4, \mathbf{g}_4, x_4), x_3 = x_4,$$
$$\mathrm{HasSecLevel}(x_4, \mathbf{secret}), \mathrm{Includes}(x_0, x_4)\}.$$

Then, Q_1 and Q_2 could be defined accordingly.

$$Q_1 = \mathrm{Call}(t_1, \mathbf{g}_1, x_1) \wedge \mathrm{Call}(t_2, \mathbf{g}_2, x_2) \wedge t_1 < t_2 \wedge x_1 \% 2 = 0,$$

$$Q_2 = \mathrm{Call}(t_0, \mathbf{g}_0, x_0) \wedge \mathrm{Call}(t_3, \mathbf{g}_3, x_3) \wedge \mathrm{Call}(t_4, \mathbf{g}_4, x_4) \wedge x_3 = x_4 \wedge$$
$$\mathrm{HasSecLevel}(x_4, \mathbf{secret}) \wedge \mathrm{Includes}(x_0, x_4).$$

Let $X = \{\mathrm{Call}(2, \mathbf{g}_1, 4)\}$ and $W = \emptyset$. We know that $Q_{[t_2]} = Q_1$ and $Q_{[t_4]} = Q_2$. We then have

$$X \cup \{\mathrm{Call}(6, \mathbf{g}_2, 5)\} \cup X_G \vdash Q_{[t_2]}\big[2/t_1\big]\big[4/x_1\big]\big[6/t_2\big]\big[5/x_2\big].$$

Therefore, according to Precondition-6 we would have

$$(\mathit{callEvent}(\mathbf{g}_2, 5); e, \{\mathrm{Call}(2, \mathbf{g}_1, 4)\}, 7, \mathbb{L}, \mathcal{C}, \emptyset) \rightarrow \\ (e, \{\mathrm{Call}(2, \mathbf{g}_1, 4), \mathrm{Call}(6, \mathbf{g}_2, 5)\}, 7, \mathbb{L}, \mathcal{C}, \{\mathbf{g}_1, \mathbf{g}_2\}).$$

Since a ground form of $Q_{[t_2]}$ is derived, we add both $\mathbf{g}_1$ and $\mathbf{g}_2$ to W to ensure that we will not add any invocation information of these triggers to X, any more.

Now suppose X has grown to be

$$X = \{\mathrm{Call}(2, \mathbf{g}_1, 4), \mathrm{Call}(6, \mathbf{g}_2, 5), \mathrm{Call}(11, \mathbf{g}_3, 7), \mathrm{Call}(16, \mathbf{g}_0, [5, 7, 9])\}.$$

Note that the argument to $\mathbf{g}_0$ is a list. At time 16, $\mathrm{LoggedCall}$ is not derivable as one of triggers ($\mathbf{g}_4$) has not been called yet. We then have

$$X \cup \{\mathrm{Call}(25, \mathbf{g}_4, 7)\} \cup X_G \vdash Q_{[t_4]}\big[11/t_3\big]\big[7/x_3\big]\big[16/t_0\big]\big[[5, 7, 9]/x_0\big]\big[25/t_4\big]\big[7/x_4\big],$$

assuming that $X_G \vdash \mathrm{HasSecLevel}(7, \mathbf{secret})$. Then, according to Precondition-7 we will have

$$(\mathit{callEvent}(\mathbf{g}_4, 7); e, X, 26, \mathbb{L}, \mathcal{C}, \{\mathbf{g}_1, \mathbf{g}_2\}) \rightarrow \\ (e, X \cup \{\mathrm{Call}(25, \mathbf{g}_4, 7)\}, 26, \mathbb{L}, \mathcal{C}, \{\mathbf{g}_1, \mathbf{g}_2\}).$$

Despite the fact that a ground form of $Q_{[t_4]}$ is derived, W is not extended with trigger names, e.g., $\mathbf{g}_4$. This helps us adding invocations of these triggers to X in the future, which might help derive $\mathrm{LoggedCall}$ predicates that otherwise we were not able to derive. For instance, in this example, if $\mathrm{Call}(27, \mathbf{g}_3, 2)$, $\mathrm{Call}(31, \mathbf{g}_4, 2)$, and $\mathrm{Call}(34, \mathbf{g}_0, [1, 4, 2])$ are added later to the set of logging preconditions, $\mathrm{LoggedCall}(34, \mathbf{g}_0, [1, 4, 2])$ is derivable (assuming that $X_G \vdash \mathrm{HasSecLevel}(2, \mathbf{secret})$).

Chapter 6

Related Work and Conclusion

6.1 Related Work

Previous work by DeYoung et al. has studied audit policy specification for medical (HIPAA) and business (GLBA) processes [39, 40, 41]. 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 [22]. Their auditing system has also been implemented in a tool similar to an interactive theorem prover [29]. 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 [42], 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 [43, 30], process calculi and game theory [44]) 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 [16] and the APPLE system [45]. 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 [46], 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 [27], 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 [13] and are enforceable by security automata, as we have shown. Hence IRM rewriting techniques could be used to implement them [14]. 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. [47] 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 [8] 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 [48, 49]. Cheney [50] presents a framework for provenance, built on a notion of system traces. Recently, W3C has proposed a data model for provenance, called PROV [51], which enjoys a formal description of its specified constraints and inferences in first-order logic, [52], however the given semantics does not cover the relationship between the provenance record and the actual system behavior.

6.2 Conclusion

In this work 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. To demonstrate, we define a prototype rewriting algorithm for a functional calculus that instruments a class of logging specifications defined in first order logic, and prove the algorithm correct.

Bibliography

[1] Butler W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, 2004.

[2] Dean Povey. Optimistic security: A new access control paradigm. In *Proceedings of the 1999 Workshop on New Security Paradigms*, pages 40–45, 1999.

[3] Daniel J. Weitzner. Beyond secrecy: New privacy protection strategies for open information spaces. *IEEE Internet Computing*, 11(5):94–96, 2007.

[4] 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.

[5] Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, Ja<mes A. Hendler, and Gerald J. Sussman. Information accountability. *Communications of the ACM*, 51(6):82–87, 2008.

[6] Wen Zhang, You Chen, Thaddeus Cybulski, Daniel Fabbri, Carl A. Gunter, Patrick Lawlor, David M. Liebovitz, and Bradley Malin. Decide now or decide later? Quantifying the tradeoff between prospective and retrospective access decisions. In *Proceedings*

of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 1182–1192, 2014.

[7] Jason Tyler King, Ben Smith, and Laurie Williams. Modifying without a trace: General audit guidelines are inadequate for open-source electronic health record audit mechanisms. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, pages 305–314. ACM, 2012.

[8] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? An empirical study on logging practices in industry. In *Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, 2014.

[9] Anton Chuvakin. Beautiful log handling. In Andy Oram and John Viega, editors, *Beautiful security: Leading security experts explain how they think*. O'Reilly Media Inc., 2009.

[10] Richard A Kemmerer and Giovanni Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, 2002.

[11] Duncan Cook, Jacky Hartnett, Kevin Manderson, and Joel Scanlan. Catching spam before it arrives: Domain specific dynamic blacklists. In *Proceedings of the Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006) and the Fourth Australasian Information Security Workshop*, pages 193–202. Australian Computer Society, Inc., 2006.

[12] J. Kohlas. *Information Algebras: Generic Structures For Inference*. Discrete mathematics and theoretical computer science. Springer, 2003.

[13] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[14] Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2003.

[15] OpenMRS. `http://openmrs.org/`, 2015. Accessed: 2015-09-27.

[16] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 177–191, 2008.

[17] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. Leveraging "Choice" to automate authorization hook placement. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 145–156, October 2012.

[18] Dave King, Susmit Jha, Divya Muthukumaran, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. Automating security mediation placement. In *Proceedings of the 19th European Symposium on Programming*, pages 327–344, 2010.

[19] Peter Pharow and Bernd Blobel. Mobile health requires mobile security: Challenges, solutions, and standardization. *Studies in Health Technology and Informatics*, 136:697–702, 2008.

[20] David Kotz, Sasikanth Avancha, and Amit Baxi. A privacy framework for mobile health and home-care systems. In *Proceedings of the First ACM Workshop on Security and Privacy in Medical and Home-care Systems*, pages 1–12. ACM, 2009.

[21] Pam Matthews and Holly Gaebel. Break the glass. In *HIE Topic Series*. Healthcare Information and Management Systems Society, 2009. `http://www.himss.org/files/himssorg/content/files/090909breaktheglass.pdf`.

[22] Anupam Datta, Jeremiah Blocki, Nicolas Christin, Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kirli Kaynar, and Arunesh Sinha. Understanding and protecting privacy: For-

mal semantics and principled audit mechanisms. In *Proceedings of the 7th International Conference on Information Systems Security*, pages 1–27, 2011.

[23] Juerg Kohlas and Juerg Schmid. An algebraic theory of information: An introduction and survey. *Information*, 5(2):219–254, 2014.

[24] Usage statistics module. `https://wiki.openmrs.org/display/docs/Usage+Statistics+Module`, 2010. Accessed: 2015-09-27.

[25] Debmalya Biswas and Valtteri Niemi. Transforming privacy policies to auditing specifications. In *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, pages 368–375, 2011.

[26] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.

[27] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364, 2005.

[28] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 1105–1112, New York, NY, USA, 2006. ACM.

[29] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: The-

ory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 151–162, 2011.

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

[31] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 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.

[32] Syed Zain Rizvi, Philip W. L. Fong, Jason Crampton, and James Sellwood. Relationship-based access control for an open-source medical records system. In *ACM Symposium on Access Control Models and Technologies*, 2015.

[33] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Coq formalization of auditing correctness for core functional calculus. `https://github.com/sepehram/auditing-instrumentation-correctness`, 2015.

[34] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[35] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Retrospective Security Module for OpenMRS. `https://github.com/sepehram/retro-security-openmrs`, 2015.

[36] Spring framework. `http://projects.spring.io/spring-framework/`, 2015. Accessed: 2015-09-27.

[37] XSB. `http://xsb.sourceforge.net/`, 2012. Accessed: 2015-09-27.

[38] Logic for your app. `http://interprolog.com/`, 2014. Accessed: 2015-09-27.

[39] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kirli Kaynar, and Anupam Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *Proceedings of the 2010 ACM Workshop on Privacy in the Electronic Society*, pages 73–82, 2010.

[40] Henry DeYoung, Deepak Garg, Dilsun Kaynar, and Anupam Datta. Logical specification of the GLBA and HIPAA privacy laws. Technical Report CMU-CyLab-10-007, Carnegie Mellon University, April 2010.

[41] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. Privacy policy specification and audit in a fixed-point logic: How to enforce HIPAA, GLBA, and all that. Technical Report CMU-CyLab-10-008, Carnegie Mellon University, April 2010.

[42] Benjamin Böck, David Huemer, and A. Min Tjoa. Towards more trustable log files for digital forensics by means of "trusted computing". In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA '10, pages 1020–1027, Washington, DC, USA, 2010. IEEE Computer Society.

[43] Ricardo Corin, Sandro Etalle, J. I. den Hartog, Gabriele Lenzini, and I. Staicu. A logic for auditing accountability in decentralized systems. In *Formal Aspects in Security and Trust*, pages 187–201, 2004.

[44] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Towards a theory of accountability and audit. In *Proceedings of the 14th European Symposium on Research in Computer Security*, pages 152–167, 2009.

[45] Sandro Etalle and William H. Winsborough. A posteriori compliance control. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 11–20, 2007.

[46] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383. ACM, 2005.

[47] Nataliya Guts, Cédric Fournet, and Francesco Zappa Nardelli. Reliable evidence: Auditability by typing. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 168–183, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. *Lecture Notes in Mathematics - Springer Verlag*, pages 316–330, 2000.

[49] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 539 – 550, 2006.

[50] James Cheney. A formal framework for provenance security. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium*, pages 281–293, 2011.

[51] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV data model. `http://www.w3.org/TR/2013/REC-prov-dm-20130430`, 2013. Accessed: 2015-02-07.

[52] James Cheney. Semantics of the PROV data model. `http://www.w3.org/TR/2013/NOTE-prov-sem-20130430`, 2013. Accessed: 2015-02-07.