Correct Audit Logging in Concurrent Systems (Technical Report)

Sepehr Amir-Mohammadian[a,1] Chadi Kari[a,2]

[a] *School of Engineering and Computer Science, University of the Pacific, Stockton, California 95211*

Abstract

Audit logging provides post-facto analysis of runtime behavior for different purposes, including error detection, amelioration of system operations, and the establishment of security in depth. This necessitates some level of assurance on the quality of the generated audit logs, i.e., how well the audit log represents the events transpired during the execution. Information-algebraic techniques have been proposed to formally specify this relation and provide a framework to study correct audit log generation in a provable fashion. However, previous work fall short on how to guarantee this property of audit logging in concurrent environments. In this paper, we study an implementation model in a concurrent environment. We propose an algorithm that instruments a concurrent system according to a formal specification of audit logging requirements, so that any instrumented concurrent system guarantees correct audit log generation. As an application, we consider systems with microservices architecture, where logging an event by a microservice is conditioned on the occurrence of a collection of events that take place in other microservices of the system.

Keywords: Audit logging, Concurrent systems, Programming languages, Security

1 Introduction

Reliable audit logging is essential to provide secure computation through the after-the-fact analysis of the audit log. Audit logging is used along with preventive security mechanisms to enable in-depth enforcement of security. In-depth enforcement of security refers to multiple layers of pre-execution, runtime, and post-execution techniques to ensure the legitimacy of the computation. Examples abound, e.g., a medical records system that enforces preventive measures including user authentication, static/dynamic information flow analysis to prevent leakage or corruption of data [1], and access authorization, e.g., to deny illegitimate access of certain users to certain medical data. Moreover, the system engages in the collection and a posteriori analysis of audit logs for different purposes, including the satisfaction of accountability goals, e.g., established by Health Insurance Portability and Accountability Act (HIPAA) [2,3], reinforcement of access control [4,5], etc.

Using audit logging along with preventive mechanisms has two major applications: 1) Post-execution analysis of audit logs provides a platform to detect security violations based on the logged evidence [6,7]. This class of logging policies rely on the notions of user accountability and deterrence. 2) Audit logging is used to detect existing vulnerabilities in the preventive security mechanisms and ameliorate those mechanisms [8,9].

In both of the aforementioned applications, effectiveness of in-depth security relies on the correctness and efficiency of the generated audit log and its after-the-fact analysis. Audit log correctness and efficiency reflect on some challenges in the generation of audit logs. Correct audit logging must record factual information about the runtime behavior, which may be ensured by the verification of auditing policies and their runtime enforcement mechanisms. Moreover, a correct audit log must include sufficient information according to what the auditing policy specifies. In addition, efficiency of audit logging must be emphasized in in-depth security in order to improve system performance regarding the collection and analysis of audit logs. Efficient audit logging entails to only record necessary information about the computation, rather than naively collecting *all* events

[1] Email: samirmohammadian@pacific.edu

[2] Email: celkari@pacific.edu

in the log. These issues have been challenging in the wild, for instance resulting in failure to safeguard against data breaches, and are considered as one of the top ten most critical security risks by Open Web Application Security Project [10].

To establish a formal foundation for audit logging, a general semantics of audit logs has been defined [7,11] using the theory of *information algebra* [12]. This line of work helps to study whether the mechanism of enforcing an audit logging policy is correct and efficient on the basis of the proposed information-algebraic framework. Both program execution traces and audit logs are interpreted as information elements in this framework. In essence, the relation between the information in execution traces and the information in audit logs is formulated according to the established notion of information containment. An audit log is defined correct if it satisfies this relation. This formulation facilitates the separation of the specification of auditing requirements from programs, which is of great value in practice. This way, rather than manual inlining of audit logging in the code, algorithms can be proposed that automatically instrument the code with audit logging capabilities. The semantic framework enables algorithms for implementing general classes of specifications for auditing and establish conditions that guarantee the enforcement of those specifications by such algorithms.

The aforementioned line of work relies on the proposed semantic framework for audit log generation whose implementation model is constrained to linear process executions. This limitation, in practice, restricts the application of the framework to systems where a single thread of execution is involved in the generation of audit logs. For instance, in the case study of a medical records system [11], audit logging capability is considered as an extension to the web server program, and all preconditions for logging depend on the events that transpire in the same program execution thread. As an example consider breaking the glass event [13]. Breaking the glass is used in critical situations to bypass access control. By breaking the glass, system users increase their authority in the system in order to gain access to certain data, but simultaneously admit to be accountable for their actions. Breaking the glass event is a precondition to log accesses to particular patient information. Instrumentation of medical records web server guarantees correct audit logging as long as such events occur in the execution trace of the single-threaded web server. This eliminates the possibility of distributing authentication and authorization tasks to other concurrent components of the system. Such restriction encourages us to study the semantics of audit logging in concurrent environments that underlies correct instrumentation of multithreaded and multiprocess applications for auditing purposes. Indeed, real-world examples of inadequate logging and monitoring in concurrent and distributed systems, e.g., a recent security incident in a retailer's network of POS systems [14], demonstrates how crucial it is to ensure the correctness of audit logging mechanisms.

The proposed semantic framework needs to provide a mechanism to specify auditing requirements based on concurrent execution traces. Our framework needs to be general enough to encompass different audit log generation and representation approaches as its instances. The generality of information theoretic models have already been shown in this realm [11]. We demonstrate that such a model can be used for concurrent systems. We use the model to interpret audit logs, specify audit logging requirements and define correct enforcement of such requirements in concurrent systems. Similar to the previous work on linear process executions, correctness of log in concurrent environments is conditioned on the specifications of auditing requirements through the comparison between the information contained in the log and the information advertised by those specifications.

We instantiate our general model with a sufficiently expressive language in order to specify and enforce auditing requirements in concurrent environments. Horn clause logic is a proper language for this purpose due to straightforward modeling of execution traces as sets of facts, sufficient expressivity to specify auditing requirements and available logic programming implementations, e.g., [15,16].

A formal language model is used to specify and establish correct enforcement of audit logging policies in concurrent systems according to the developed framework. We use a variant of π-calculus [17] with unlabeled reduction semantics for this purpose. This formalism provides a model for developing tools with correctness guarantees. This model enjoys the following features.

- Concurrency: In order to specify multithreaded programs and multiprocessor systems, the language model supports concurrent process executions with interprocess communication (IPC) through message passing.
- Generality: While different process calculi are potential choices to describe our implementation model, we use a variant of π-calculus due to its sufficiently concise and high level syntax and semantics to describe interactions among processes. This facilitates formulation of a wide range of concurrent systems.
- Timing: We need to be able to specify the ordering of interesting events for the sake of specifying auditing requirements. For example, in break the glass policies access to particular patient information is logged as long as the glass is already broken. In order to implement such specifications, we need to apply a timing mechanism that is shared among all processes of the system. Each step of concurrent execution of processes updates this universal time.
- Named functions: To specify auditing requirements, a fundamental unit of secure operations is required. Functions can be considered as abstractions of these fundamental units in different languages and systems.

Our language model supports named functions, in terms of sub-agents of each agent.

Using the formalism with aforementioned features enables us to model concurrent environments that guarantee the correct generation of audit logs according to the developed semantic framework. In this paper, we propose an instrumentation algorithm that receives a concurrent system as input and modifies the system according to a precise specification of audit logging requirements. We show that this algorithm is correct (based on the semantic framework), and hence the instrumented concurrent system generates correct audit logs. As implied earlier, enforcement of audit logging policies through code instrumentation separates policy from code, provides a foundation to study the effectiveness of enforcement mechanism using formal methods, and can be applied automatically to legacy code to enhance system accountability.

Since our model is based on process calculi, IPC is handled by message-passing. Modeling alternative IPC approaches for correct audit logging, e.g., shared memory and/or files, is considered as potential future work.

Case studies that benefit from the result of this work include deployment of correct logging capabilities in multiprocess and multithreaded client-server and peer-to-peer applications, microservices, etc. While this paper provides a prototype instrumentation algorithm in abstract settings, as a future work, we aim to deploy our existing instrumentation algorithm in Spring Boot [18], a Java microservices framework, that facilitates code instrumentation through aspect-oriented programming [19].

1.1 Paper Outline

The rest of the paper is organized as follows. In Section 1.2, we discuss an illustrative example for audit logging in concurrent systems and in Section 1.3 we discuss the threat model. Section 2 reviews the information-algebraic semantics of audit logging and instantiation of the model with first-order predicate logic. Section 3 discusses the implementation model in detail. In particular, Section 3.1 introduces the syntax and semantics of the source system, a variant of π-calculus. In Section 3.2, we study a class of logging specifications that assert temporal relations among function invocations, potentially in different concurrent components of a system. Section 3.3 studies the syntax and semantics of the target system. In Section 3.4, we propose our instrumentation algorithm, along with the properties of interest that the algorithm satisfies. In Section 4, related work is discussed. Finally, Section 5 concludes the paper and specifies future work.

1.2 An Example: Microservices-based Medical Records Systems

In this section, an oversimplified example is given that illustrates the application of audit logging in concurrent environments. We will revisit this example later in the paper (through Examples 3.1 and 3.2) to explain sample instantiations of our formal framework.

Many applications have been shifting their architecture from a traditional monolithic structure toward service-oriented architecture (SOA) in order to boost maintainability, continuous deployment and testing, adaptation to new technologies, system security, fault tolerance, etc. One popular deployment approach to SOA is where an application is decomposed to a set of highly collaborative processes, called microservices. A microservice must be minimal, independent, and fine-grained. Minimality constrains a microservice to access and manipulate certain data types within an application, ideally a single database per each service. Microservice instances run independently in their own containers, virtual machines, or hosts. To accomplish its own goals, a microservice communicates with other microservices of the application through message passing, or remote procedure calls (RPCs). Jolie [20] is the programming language for developing applications with microservices architecture. Its formal semantics [21,22] is defined as a process calculus, inspired by π-calculus.

The need to better streamline healthcare services is pushing medical record systems toward microservices [23]. In fact, a new study shows that microservices-based healthcare is anticipated to experience fivefold increase in market value within the next few years [24].

In what follows we describe a simple example of microservices-based medical records system, where audit logging for certain events is necessary, as dictated by the accountability requirements. Figure 1 depicts a medical records system with microservices architecture that includes Authorization and Patient services (among others). Application front-end includes API gateway that multiplexes user requests (from different clients, e.g., web, mobile applications, etc.) to certain microservices. Patient microservice manages the information about patients, e.g., their medical history. Authorization microservice handles different operations to authorize access to system data, including e.g., breaking the glass.

As mentioned earlier, by breaking the glass, the user agrees to comply with accountability regulations. The common solution to follow accountability regulations is to generate trail of audit logs at runtime. One such audit logging requirement may be as follows: "Any attempt by a healthcare provider to read patient medical data must be stored in the log, if that provider has already broken the glass."

This example demonstrates the core ideas that we are pursuing in this paper:

- A concurrent system is employed, where each component runs independently and in collaboration with other

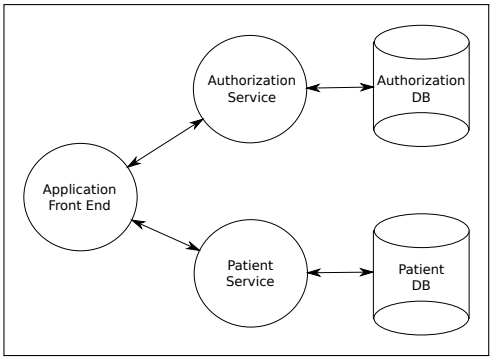

Fig. 1. Authorization and Patient microservices of a medical records system.

components, e.g., a medical records system with different microservices including the ones described above.

- Audit logging requirements necessitate logging certain events in a concurrent component, provided that a set of other events have previously occurred in potentially other concurrent components, e.g., logging the event of reading medical history in Patient microservice if the glass is already broken in Authorization microservice.
- We investigate an algorithmic approach to establish correct audit logging for concurrent environments according to the already-established audit logging requirements. We expect correctness of audit logging in our medical records system example, for instance, to imply only logging the reading attempts by the user who has broken the glass. This avoids missing any logging event, as well as logging unnecessary events. We accomplish this by instrumenting Authorization and Patient microservices, and in particular the operations of interest, i.e., breaking the glass and reading patient medical history.

1.3 Threat Model

We assume that the concurrent system subjected to instrumentation is not supporting audit logging in the first place, or is suffering from either insufficient or overzealous audit log generation. However, we assume that the concurrent system that is deployed according to our implementation model, passes both static and dynamic checks, e.g., syntactic checks, type checks, compilation, interpretation, etc. We trust the compiler/interpreter, and the runtime environment in which that system is being executed. Moreover, we trust the implementation of our instrumentation algorithm, and its compilation and/or interpretation, along with the runtime environment in which the instrumentation algorithm is executed. We also trust the integrity of logging specifications that assert audit logging requirements. Finally, we trust the compilation/interpretation process for the instrumented concurrent system that is deployed based on our implementation model, as well as the system's runtime environment. Security of the messages transmitted between concurrent components and the generated audit logs is considered to be out-of-scope.

These assumptions help us to purposefully focus on the essence of logging, i.e., whether logs are generated correctly in the first place and independent of external concerns including reliability of the underlying execution and communication system, latency, etc. which are explored in related work (Section 4).

2 Semantics of Audit Logging

In order to provide a standalone formal presentation, in this section we review the information-algebraic semantics of audit logging and the instantiation of the semantic framework with first-order logic, which is originally proposed by Amir-Mohammadian et al. [11]. We have applied minor modifications to the model to better suit concurrency and nondeterministic runtime behavior, inherent to concurrent systems.

2.1 Information-Algebraic Semantic Framework

In order to specify how audit logs are generated at runtime, we need to abstract system states and their evolution through the computation. A *system configuration* κ abstracts the state of the system at a given point during the execution. Let $\mathcal{K}$ denote the set of system configurations. We posit a binary reduction relation among configurations, i.e., $(\longrightarrow) \subseteq \mathcal{K} \times \mathcal{K}$ which denotes the computational steps, and is used in the standard infix form.[3] A *system trace* τ is a potentially infinite sequence of system configurations, i.e.,

[3] A notational convention throughout the paper is that infix operators and relations are wrapped with parentheses when their signature are specified.

$\tau = \kappa_0\kappa_1\cdots$, where κ_i is the ith configuration in sequence, and $\kappa_i \longrightarrow \kappa_{i+1}$. We denote the set of all traces by $\mathcal{T}$, and define *prefix*(τ) as the set of all prefixes of τ.

Information algebra is used to define the notion of correctness for audit logs. In Section 2.2, we instantiate this abstract algebraic structure to model a specific class of audit logging requirements. We define an information algebra in the following.

Definition 2.1 (Information algebra) *An information algebra (Φ, Ψ) is a two-sorted algebra consisting of an Abelian semigroup of information elements, Φ, as well as a lattice of querying domains, Ψ. Two fundamental operators are presumed in this algebra: a combination operator, $(\otimes) : \Phi \times \Phi \to \Phi$, and a focusing operator, $(^{\Rightarrow}) : \Phi \times \Psi \to \Phi$. An Information algebra (Φ, Ψ) satisfies a set of properties, in connection to combination and focusing operators.*[4] *We let $X, Y, Z, \cdots$ to range over elements of Φ, and E range over Ψ.*

$X, Y \in \Phi$ are information elements that can be combined to make a more inclusive information element $X \otimes Y$. $E \in \Psi$ is a querying domain with a certain level of granularity that is used by the focusing operator to extract information from an information element X, denoted by $X^{\Rightarrow E}$. For example, relational algebra is an instance of information algebra, in which relations instantiate information elements, sets of attributes instantiate querying domains, natural join of two relations defines the combination operator, and projection of a relation on a set of attributes defines the focusing operator [12].

Combination of information elements induces a partial order relation $(\preccurlyeq) \subseteq \Phi \times \Phi$ among information elements, defined as follows: $X \preccurlyeq Y$ iff $X \otimes Y = Y$. Intuitively, $X \preccurlyeq Y$ means that Y contains the information element X.

As part of the semantics of audit logging, we treat execution traces as information elements, i.e., the information content of the execution trace. To this end, we posit $\lfloor\cdot\rfloor : \mathcal{T} \to \Phi$ as a mapping in which, intuitively, $\lfloor\tau\rfloor$ refers to the information content of the trace τ. We also impose the condition that $\lfloor\cdot\rfloor$ be injective and monotonically increasing, i.e., if $\tau' \in$ *prefix*(τ) then $\lfloor\tau'\rfloor \preccurlyeq \lfloor\tau\rfloor$. This ensure that as the execution trace grows in length, it contains more information.

In the following definition, we define audit logging requirements in an abstract form. We call this abstraction a *logging specification*. This definition is abstract enough to encompass different execution models, as well as different representations of information. In Sections 2.2 and 3.2, we instantiate this definition with a more concrete structure that guides us on how to implement audit logging requirements.

Definition 2.2 (Logging specifications) *Logging specification LS is defined as a mapping from system traces to information elements, i.e., $LS : \mathcal{T} \to \Phi$. Intuitively, $LS(\tau)$ declares what information must be logged, if the system follows the execution trace τ.*

Note that even though $\lfloor\cdot\rfloor$ and LS have the same signature, i.e., maps from traces to information elements, they are conceptually different. $\lfloor\tau\rfloor$ is the whole information contained in τ, whereas $LS(\tau)$ is the information that is supposed to be recorded in the log, if the system follows the execution trace τ.

We denote an audit log with $\mathbb{L}$ which represents a set of data, gathered at runtime. Let $\mathcal{L}$ denote the set of audit logs. In order to judge about the correctness of an audit log, the information content of the audit log needs to be studied in comparison to the information content of the trace that generates that audit log. To this end, we define a mapping that returns the information content of an audit log. We abuse the notation and consider $\lfloor\cdot\rfloor : \mathcal{L} \to \Phi$ as such mapping. Therefore, $\lfloor\mathbb{L}\rfloor$ refers to the information content of the audit log $\mathbb{L}$. We assume that $\lfloor\cdot\rfloor$ on audit logs is injective and monotonically increasing, i.e., if $\mathbb{L} \subseteq \mathbb{L}'$ then $\lfloor\mathbb{L}\rfloor \preccurlyeq \lfloor\mathbb{L}'\rfloor$. Therefore, the more inclusive the audit log is, it contains more information.

The notion of correct audit logging can be defined based on an execution trace and a logging specification. To this end, the information content of the audit log is compared to the information that the logging specification dictates to be recorded in the log, given the execution trace. The following definition captures this relation.

Definition 2.3 (Correctness of audit logs) *Audit log $\mathbb{L}$ is* correct *wrt a logging specification LS and a system trace τ iff both $\lfloor\mathbb{L}\rfloor \preccurlyeq LS(\tau)$ and $LS(\tau) \preccurlyeq \lfloor\mathbb{L}\rfloor$ hold. The former refers to the necessity of the information in the audit log, and the latter refers to the sufficiency of those information.*

A system that generates audit logs at runtime includes the stored logs as part of its configuration. Let the mapping *logof* $: \mathcal{K} \to \mathcal{L}$ denote the residual log of a given system configuration, i.e., *logof*(κ) is the set of all recorded audit logs in configuration κ. It is natural to assume that the residual log within configurations grows larger as the execution proceeds. The residual log of a trace is then defined using *logof*.

Definition 2.4 (Residual log of a system trace) *The residual log of a finite system trace τ is $\mathbb{L}$, denoted by $\tau \rightsquigarrow \mathbb{L}$, iff $\tau = \kappa_0\kappa_1\cdots\kappa_n$ and logof$(\kappa_n) = \mathbb{L}$.*

[4] We avoid discussing these properties in detail here for the sake of brevity. Readers are referred to [12] for the complete formulation.

Note that if $\tau \rightsquigarrow \mathbb{L}$, then $\mathbb{L}$ is not necessarily correct wrt a given LS and a trace τ. If the residual log of a trace is correct throughout the execution, then that trace is called *ideally-instrumented*. System trace τ is ideally instrumented for a logging specification LS iff for any trace τ' and audit log $\mathbb{L}$, if $\tau' \in \mathit{prefix}(\tau)$ and $\tau' \rightsquigarrow \mathbb{L}$ then $\mathbb{L}$ is correct wrt τ' and LS. Indeed audit logging is an enforceable security property on a trace of execution [11]. Given a logging specification, ideally-instrumented traces induce a safety property [25], and hence implementable by inlined reference monitors [26], and edit automata [27].

Let $\mathfrak{s}$ be a concurrent system with an operational semantics. $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ can produce trace τ', either deterministically or non-deterministically, and $\tau \in \mathit{prefix}(\tau')$. We abuse the notation and use $\kappa \Downarrow \tau$ to denote the same concept for configuration κ. We follow program instrumentation techniques, in order to enforce a logging specification on a system. An *instrumentation algorithm* receives the concurrent system as input along with the logging specification, and instruments the system with audit logging capabilities so that the instrumented system generates the required "appropriate" log. An instrumentation algorithm is a partial function $\mathcal{I} : (\mathfrak{s}, LS) \mapsto \mathfrak{s}'$ that instruments $\mathfrak{s}$ according to LS aiming to generate audit logs appropriate for LS. We call $\mathfrak{s}$ the *source system*, and the instrumented system, i.e., $\mathcal{I}(\mathfrak{s}, LS) = \mathfrak{s}'$, the *target system*. Source and target traces refer to the traces of the source and target systems, resp.

It is natural to expect that the instrumentation algorithm would not modify the semantics of the original system drastically. The target system must behave roughly similar to the source system, except for the operations related to audit logging. We call this attribute of an instrumentation algorithm *semantics preservation*, and define it in the following. This definition is abstract enough to encompass different source and target systems (with different runtime semantics), and instrumentation techniques. The abstraction relies on a binary relation $:\approx$, called *correspondence relation*, that relates the source and target traces. Based on different implementations of the source and target systems, and the instrumentation algorithm, the correspondence relation can be defined accordingly.

Definition 2.5 (Semantics preservation by the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is semantics preserving iff for all systems $\mathfrak{s}$ and logging specifications LS, where $\mathcal{I}(\mathfrak{s}, LS)$ is defined, the following conditions hold: 1) For any trace τ, if $\mathfrak{s} \Downarrow \tau$, then there exists some trace τ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, and $\tau :\approx \tau'$. 2) For any trace τ, if $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau$, then there exists some trace τ' such that $\mathfrak{s} \Downarrow \tau'$, and $\tau' :\approx \tau$.*

Another property of the instrumentation algorithm is to ensure that it is *deadlock-free*, meaning that instrumenting a system does not introduce new states being stuck. Let source system $\mathfrak{s}$ generate trace τ, and $\mathcal{I}(\mathfrak{s}, LS)$ generate trace τ' such that $\tau :\approx \tau'$. Then, we call $\mathcal{I}(\mathfrak{s}, LS)$ being stuck if $\mathfrak{s}$ can continue execution following τ (at least for one extra step), while $\mathcal{I}(\mathfrak{s}, LS)$ cannot continue execution following τ'.

Definition 2.6 (Deadlock-freeness of the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is deadlock-free iff for any source system $\mathfrak{s}$, logging specification LS, traces τ and τ', and configuration κ, if $\mathfrak{s} \Downarrow \tau$, $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\mathfrak{s} \Downarrow \tau\kappa$, then there exists some configuration κ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'\kappa'$.*

Besides these properties, another important feature of an instrumentation algorithm is the quality of audit logs generated by the instrumented system. The information-algebraic semantic framework provides a platform to define correct instrumentation algorithms for audit logging purposes. Let $\mathfrak{s}$ be a target system, and τ be a source trace. Simulated logs of τ by $\mathfrak{s}$ is the set $\mathit{simlogs}(\mathfrak{s}, \tau)$ defined as $\mathit{simlogs}(\mathfrak{s}, \tau) = \{\mathbb{L} \mid \exists \tau'.\mathfrak{s} \Downarrow \tau' \wedge \tau :\approx \tau' \wedge \tau' \rightsquigarrow \mathbb{L}\}$. Using this set, we can define correctness of instrumentation algorithms in a straightforward manner. Intuitively, the instrumentation algorithm $\mathcal{I}$ is correct if the instrumented system generates audit logs that are correct wrt the logging specification and the source trace. This must hold for any source system, any logging specification, and any possible log generated by the instrumented system.

Definition 2.7 (Correctness of the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is correct iff for all source systems $\mathfrak{s}$, traces τ, and logging specifications LS, $\mathfrak{s} \Downarrow \tau$ implies that for any $\mathbb{L} \in \mathit{simlogs}(\mathcal{I}(\mathfrak{s}, LS), \tau)$, $\mathbb{L}$ is correct wrt LS and τ.*

2.2 Instantiation of Logging Specification

In Definition 2.2, logging specification is defined abstractly as a mapping from system traces to information elements. For a more concrete setting, this definition needs to be instantiated with appropriate structures in a way that is useful in the deployment of audit logging. In essence, we need to instantiate information algebra (Definition 2.1). We are interested in logical specification of audit logging requirements due to its easiness of use, expressivity power, well-understood semantics, and off-the-self logic programming engines for subsets of first-order logic (FOL), e.g., Horn clause logic. To this end, in this section, we instantiate information algebra with FOL, which is expressive enough to specify computational events, and the temporal relation among them. Indeed, other variants of logic may also be considered for this purpose.

In order to instantiate information algebra, it is required to specify the contents of the set of information elements Φ and the lattice of querying domains Ψ, along with the definitions of combination and focusing operators. Definitions 2.8, 2.9, and 2.10 accomplish these instantiations.

Definition 2.8 instantiates an FOL-based set of information elements. An information element in our instantiation is a closed set of FOL formulas, under a proof-theoretic deductive system.

Definition 2.8 (Set of closed sets of FOL formulas) *Let φ range over FOL formulas, and Γ range over sets of FOL formulas. $\Gamma \vdash \varphi$ denotes a judgment derived by a sound and complete natural deduction proof theory of FOL. We define closure operation Closure as $Closure(\Gamma) = \{\varphi \mid \Gamma \vdash \varphi\}$. Then, the set of closed set of FOL formulas is defined as $\Phi_{FOL} = \{\Gamma \mid \Gamma = Closure(\Gamma)\}$.*

Definition 2.9 instantiates the lattice of querying domains for the FOL-based information algebra. A query domain is a subset of FOL, defined over certain predicate symbols.

Definition 2.9 (Lattice of FOL sublanguages) *Let Preds be the set of all assumed predicate symbols along with their arities. If $S \subseteq Preds$, then we denote the sublanguage $FOL(S)$ as the set of well-formed FOL formulas over predicate symbols in S. The set of all such sublanguages $\Psi_{FOL} = \{FOL(S) \mid S \subseteq Preds\}$ is a lattice induced by set containment relation.*

Lastly, Definition 2.10 instantiates the combination and focusing operators for the FOL-based information algebra. Combination is the closure of the union of two sets of formulas. Focusing is the closure of the intersection of an information element and a query domain.

Definition 2.10 (Combination and focusing in (Φ_{FOL}, Ψ_{FOL})) *Let $(\otimes) : \Phi_{FOL} \times \Phi_{FOL} \to \Phi_{FOL}$ be defined as $\Gamma \otimes \Gamma' = Closure(\Gamma \cup \Gamma')$, and $(\stackrel{\Rightarrow}{}) : \Phi_{FOL} \times \Psi_{FOL} \to \Phi_{FOL}$ be defined as $\Gamma^{\Rightarrow FOL(S)} = Closure(\Gamma \cap FOL(S))$.*

(Φ_{FOL}, Ψ_{FOL}) is an information algebra, given the Definitions 2.8, 2.9, and 2.10.[5] In order to use (Φ_{FOL}, Ψ_{FOL}) as a framework for audit logging, we also need to instantiate the mapping $\lfloor\cdot\rfloor$, introduced in Section 2.1, to interpret both execution traces and audit logs as information elements.

Definition 2.11 (Mapping traces and audit logs to information elements in (Φ_{FOL}, Ψ_{FOL})) *Let $toFOL(\cdot) : (\mathcal{T} \cup \mathcal{L}) \to FOL(Preds)$ be an injective and monotonically increasing function. Then, we instantiate $\lfloor\cdot\rfloor = Closure(toFOL(\cdot))$ in order to interpret both traces and logs as information elements in (Φ_{FOL}, Ψ_{FOL}).*

Now we can instantiate logging specification LS in the information algebra (Φ_{FOL}, Ψ_{FOL}). To this end, a set of audit logging rules and definitions are assumed to be given in FOL. Let Γ be this set. Moreover, a set of predicate symbols are assumed that reflect on the predicates whose derivation need to be logged at runtime. This set is denoted by S. A logging specification in this setting, receives a trace τ, combines the information content of τ with closure of Γ, and then focuses on the predicates specified in S. Intuitively, given Γ and S, a logging specification maps a trace τ to the set of all predicates whose symbols are in S, and are derivable given rules in Γ and the events in τ.

Definition 2.12 (Logging Specification in (Φ_{FOL}, Ψ_{FOL})) *Given a set of FOL formulas Γ and a subset of predicate symbols $S \subseteq Preds$, a logging specification $spec(\Gamma, S) : \mathcal{T} \to \Phi_{FOL}$ is defined as $spec(\Gamma, S) = \tau \mapsto (\lfloor\tau\rfloor \otimes Closure(\Gamma))^{\Rightarrow FOL(S)}$.*

3 Implementation Model on Concurrent Systems

In this section, we propose an implementation model for correct audit logging in concurrent systems. To this end, we use a variant of π-calculus to specify the concurrent system, and propose an instrumentation algorithm that retrofits the system according to a given logging specification. We then specify and prove the properties of interest, including the correctness of the instrumentation algorithm (Definition 2.7).

In Section 3.1, the syntax and semantics of the source system model is introduced. Section 3.2 proposes a class of logging specifications that can specify temporal relations among computational events in concurrent systems. Section 3.3 describes the syntax and semantics of the systems enhanced with audit logging capabilities. Lastly, in Section 3.4, we discuss the instrumentation algorithm and the properties it satisfies.

3.1 Source System Model

We consider a core π-calculus as our source concurrent system model, denoted by Π. One major distinguishing feature of π-calculus is modeling mobile processes using the same category of names for both links and transferable objects, along with scope extrusion. However, mobility is not used in our implementation model. Therefore other seminal process calculi e.g., CSP [28] and CCS [29] can also be considered for this purpose. We employ π-calculus due to its concise syntax and simple semantics that provides a clean and sufficiently abstract specification of the required interactions among concurrent components of the system. The syntax

[5] The reader is referred to [27] for the detailed proof.

and semantics of the source system are defined in the following. It is based on the representation of the calculus given in [30] which deviates from standard π-calculus by dropping silent prefixes, unguarded summations and labeled reduction system, for the sake of simplicity and conciseness.

3.1.1 Syntax

Let $\mathcal{N}$ be the infinite denumerable set of names, and $a, b, c, \cdots$ and $x, y, z, \cdots$ range over them.

Prefixes Prefixes α are defined as $\alpha ::= a(x) \mid \bar{a}x$. Prefix $a(x)$ is the input prefix, used to receive some name with placeholder x on link a. Prefix $\bar{a}x$ is the output prefix, used to output name x on link a.

Agents and sub-agents Let A, B, C, D range over agent and sub-agent names, and $\mathcal{A}$ be the finite set of such names. Agents (processes) and sub-agents (subprocesses) P are defined as: $P ::= \mathbf{0} \mid \alpha.P \mid (P|P) \mid (\nu x)P \mid C(y_1, \cdots, y_n)$. $\mathbf{0}$ refers to the nil process. $\alpha.P$ provides a sequence of operations in the process; first input/output prefix α takes place, and then P executes. $P|P$ provides parallelism in the system. $(\nu x)P$ restricts (binds) name x within P. $C(y_1, \cdots, y_n)$ refers to the (sub-)agent invocation C with parameters $y_1, \cdots, y_n$. Let P, Q, R range over processes and subprocesses.

Free and bound names Name restriction and input prefix bind names in a process. We denote the set of free names in process P with $fn(P)$. α-conversion for bound names is defined in the standard way.

Notational conventions A sequence of names is denoted by $\tilde{a}$, i.e., $a_1, \cdots, a_l$ for some l. A sequence of name restrictions in a process $(\nu a_1)(\nu a_2)\cdots(\nu a_l)P$ is shown by $(\nu a_1 a_2 \cdots a_l)P$, or in short $(\nu\tilde{a})P$. We skip specifying the input name, if it is not free in the following process, i.e., $a.P$ refers to $a(x).P$ where $x \notin fn(P)$. $\bar{a}.P$ refers to outputting a value on link a that can be elided, e.g., due to lack of relevance in discussion.

Codebases Agent definitions are of the form $A(x_1, \cdots, x_n) \triangleq P$. Let's denote the set of agent and sub-agent definitions with $\mathcal{D}$. We assume the existence of a *universal codebase* $\mathcal{C}_\mathcal{U}$ consisting of agent definitions of such form. This codebase is used to define *top-level agents*. A top-level agent corresponds to a concurrent components of the system. Top-level agents are supposed to execute in parallel and occasionally communicate with each other to accomplish their own tasks, and in aggregate the concurrent system. Let $\mathcal{A}_\mathcal{U}$ be the set of top-level agent names such that $\mathcal{A}_\mathcal{U} \subset \mathcal{A}$. Throughout the paper we let m to be the size of $\mathcal{A}_\mathcal{U}$, comprising of $A_1, \cdots, A_m$. $\mathcal{C}_\mathcal{U}$ is defined as a function from top-level agent names to their definitions, i.e., $\mathcal{C}_\mathcal{U} : \mathcal{A}_\mathcal{U} \to \mathcal{D}$.

Moreover, we assume the existence of a *local codebase* for each top-level agent, denoted by $\mathcal{C}_\mathcal{L}(A)$ for top-level agent A. A local codebase consists of sub-agent (subprocess) definitions of the form $B^A(x_1, \cdots, x_n) \triangleq P$, where B is a sub-agent identifier, and A is a top-level agent identifier annotated in the definition of B. We treat sub-agents as internal modules or functions of a top-level process. Annotation of top-level agent identifier is used for this purpose, i.e., B^A specifies that B is a module of top-level agent A. The set of sub-agent names is denoted by $\mathcal{A}_\mathcal{L}$, defined as $\mathcal{A} - \mathcal{A}_\mathcal{U}$. $\mathcal{C}_\mathcal{L}$ is defined as the function with signature $\mathcal{C}_\mathcal{L} : \mathcal{A}_\mathcal{U} \to \mathcal{A}_\mathcal{L} \to \mathcal{D}$.

Note that any process and subprocess definition can be recursive, e.g., if $\mathcal{C}_\mathcal{U}(A) = [A(x_1, \cdots, x_n) \triangleq P]$ then $A(y_1, \cdots, y_n)$ may appear in P. In the following, we use A and B to range over top-level agent identifiers and sub-agent identifiers, resp. We use C to range over both top-level agents, A, and sub-agents, B^A. In the rest of the paper, we refer to top-level agents simply by "agents". We assume that in any (sub)process definition $C(x_1, \cdots, x_n) \triangleq P$, we have $fn(P) \subseteq \{x_1, \cdots, x_n\}$. This ensures that (sub)processes are closed.

Initial system Let $\mathcal{A}_\mathcal{U} = \{A_1, \cdots, A_m\}$. We posit a sequence of links $\tilde{c}$, that connect these agents in the system. Then the initial concurrent system $\mathfrak{s}$ is defined as

$$\mathfrak{s} ::= \langle P_\mathfrak{s}, \mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rangle, \tag{1}$$

where $P_\mathfrak{s} = (\nu\tilde{c})(A_1(\tilde{x}_1) \mid A_2(\tilde{x}_2) \mid \cdots \mid A_m(\tilde{x}_m))$, assuming $fn(P_\mathfrak{s}) = \emptyset$, i.e., $\bigcup_i \tilde{x}_i \subseteq \tilde{c}$.

Configurations We define system configurations as $\kappa ::= P$, where P is the process associated with the whole system. The initial configuration is then defined as $\kappa_0 = P_\mathfrak{s}$.

Substitutions A substitution is a function $\sigma : \mathcal{N} \to \mathcal{N}$. The notation $\{y/x\}$ is used to refer to a substitution that maps x to y, and acts as the identity function otherwise. $\{\tilde{y}/\tilde{x}\}$ is used to denote multiple explicit mappings in a substitution, where $\tilde{x}$ and $\tilde{y}$ are equal in length. $P\sigma$ refers to replacing free names in P according to σ. This is associated with renaming of bound names in P to avoid name clashes.

3.1.2 Semantics

In the following, we define evaluation contexts and the structural congruence between processes. These definitions facilitate the specification of unlabeled operational semantics in a concise manner.

Evaluation contexts A context is a process with a hole. An evaluation context $\mathcal{E}$ is a context whose hole is not under input/output prefix, i.e., $\mathcal{E} ::= [\,] \mid (\mathcal{E}|P) \mid (P|\mathcal{E}) \mid (\nu a)\mathcal{E}$.

Structural congruence Two processes P and Q are structurally congruent under the universal codebase $\mathcal{C}_\mathcal{U}$, denoted by $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$ according to the following rules.

$$\text{STRUCT}\ \frac{\mathcal{C}_{\mathcal{U}} \rhd P \equiv P' \qquad \mathcal{C}_{\mathcal{U}} \rhd Q \equiv Q' \qquad \mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P' \longrightarrow Q'} \qquad \text{CONTEXT}\ \frac{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \mathcal{E}[P] \longrightarrow \mathcal{E}[Q]} \qquad \text{CALL}\ \frac{\mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd B^A(\tilde{y}) \longrightarrow P\{\tilde{y}/\tilde{x}\}}$$

$$\text{COMM}\ \mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd a(x).P \mid \bar{a}b.Q \longrightarrow P\{b/x\} \mid Q$$

Fig. 2. Unlabeled reduction semantics of Π.

(i) Structural congruence is an equivalence relation.

(ii) Structural congruence is closed by the application of $\mathcal{E}$, i.e., $\mathcal{C}_{\mathcal{U}} \rhd P \equiv Q$ implies $\mathcal{C}_{\mathcal{U}} \rhd \mathcal{E}[P] \equiv \mathcal{E}[Q]$.

(iii) If P and Q are α-convertible, then $\mathcal{C}_{\mathcal{U}} \rhd P \equiv Q$.

(iv) The set of processes is an Abelian semigroup under $\mid$ operator and unit element $\mathbf{0}$, i.e., for any $\mathcal{C}_{\mathcal{U}}$, P, Q, and R, we have $\mathcal{C}_{\mathcal{U}} \rhd P|\mathbf{0} \equiv P$, $\mathcal{C}_{\mathcal{U}} \rhd P|Q \equiv Q|P$, and $\mathcal{C}_{\mathcal{U}} \rhd P|(Q|R) \equiv (P|Q)|R$.

(v) For all $A \in \mathcal{A}_{\mathcal{U}}$, if $\mathcal{C}_{\mathcal{U}}(A) = [A(\tilde{x}) \triangleq P]$, then $\mathcal{C}_{\mathcal{U}} \rhd A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$.

(vi) $\mathcal{C}_{\mathcal{U}} \rhd (\nu a)\mathbf{0} \equiv \mathbf{0}$.

(vii) If $a \notin fn(P)$, then $\mathcal{C}_{\mathcal{U}} \rhd (\nu a)(P|Q) \equiv P|(\nu a)Q$.

(viii) $\mathcal{C}_{\mathcal{U}} \rhd (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$.

We may elide $\mathcal{C}_{\mathcal{U}}$ in the specification of the structural congruence, if it is clear from the context.

Operational semantics We define unlabeled reduction system in Figure 2, using judgment $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \kappa \longrightarrow \kappa'$. We may elide $\mathcal{C}_{\mathcal{U}}$ and $\mathcal{C}_{\mathcal{L}}$ in the specification of reduction steps, since they are static and may be clear from the context, i.e., $\kappa \longrightarrow \kappa'$.

Note that according to structural congruence rules an agent invocation is structurally congruent to its definition (part v), and thus considered as an "implicit" step of execution according to rule STRUCT. Contrarily, rule CALL defines an "explicit" reduction step for sub-agent invocations. This is due to some technicality in our modeling: invocation of sub-agents could be logging preconditions and/or logging events (introduced in Section 3.2), and hence need special semantic treatment at the time of call (discussed later in Sections 3.3 and 3.4), e.g., deciding whether a record must be stored in the log.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1\cdots$, we use notation $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \blacktriangleright \tau$ to specify the generation of trace τ under the universal and local codebases $\mathcal{C}_{\mathcal{U}}$ and $\mathcal{C}_{\mathcal{L}}$, and according to the aforementioned unlabeled reduction system, i.e., $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

For a system trace $\tau = \kappa_0\kappa_1\cdots$, system $\mathfrak{s}$ generates τ, denoted by $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ is defined as (1), κ_0 is defined as $P_{\mathfrak{s}}$, and $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

toFOL$(\cdot)$ **instantiation for traces** In order to specify a trace logically, we need to instantiate *toFOL*$(\cdot)$ according to Definition 2.11. We consider the following predicates to logically specify a trace: Comm/3, Call/4, Context/2, UniversalCB/3, and LocalCB/4. [6]

Let $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \blacktriangleright \tau$, and $\tau = \kappa_0 \cdots \kappa\kappa' \cdots$. Moreover, let t denote a timing counter. We define a function that logically specifies a configuration within a trace. To this end, let the helper function *toFOL*(κ, t) return the logical specification of κ at time t. Essentially, *toFOL*(κ, t) specifies what the evaluation context and the redex are within κ at time t, defined as follows:

(i) *toFOL*$(\kappa, t) = \{\text{Comm}(t, a(x).P, \bar{a}b.Q), \text{Context}(t, \mathcal{E})\}$, [7] if $\kappa \equiv \mathcal{E}[a(x).P \mid \bar{a}b.Q]$ and $\kappa' \equiv \mathcal{E}[P\{b/x\} \mid Q]$.

(ii) *toFOL*$(\kappa, t) = \{\text{Call}(t, A, B, \tilde{y}), \text{Context}(t, \mathcal{E})\}$, if $\kappa \equiv \mathcal{E}[B^A(\tilde{y})]$ and $\kappa' \equiv \mathcal{E}[P\{\tilde{y}/\tilde{x}\}]$. Note that in $\text{Call}(t, A, B, \tilde{y})$, we treat $\tilde{y}$ as a single list of elements, rather than a sequence of elements passed as parameters to Call, i.e., Call is always a quaternary predicate.

As an example, consider α-converted structurally equivalent processes. Let $\kappa = a(x).(\nu b)\bar{x}b.\mathbf{0}|\bar{a}b.\mathbf{0}$. Since $\kappa \equiv \kappa' = a(x).(\nu d)\bar{x}d.\mathbf{0}|\bar{a}b.\mathbf{0}$, and $\kappa' \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$, we have $\kappa \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$ according to the rule STRUCT in Figure 2. Then, *toFOL*$(\kappa, t) =$ *toFOL*$(\kappa', t) = \{\text{Comm}(t, a(x).(\nu d)\bar{x}d.\mathbf{0}, \bar{a}b.\mathbf{0}), \text{Context}(t, [\,])\}$.

Logical specification of universal and local codebases, denoted by $\langle\mathcal{C}_{\mathcal{U}}\rangle$ and $\langle\mathcal{C}_{\mathcal{L}}\rangle$ resp., are defined as

(i) $\langle\mathcal{C}_{\mathcal{U}}\rangle = \{\text{UniversalCB}(A, \tilde{x}, P) \mid \mathcal{C}_{\mathcal{U}}(A) = [A(\tilde{x}) \triangleq P]\}$

(ii) $\langle\mathcal{C}_{\mathcal{L}}\rangle = \{\text{LocalCB}(A, B, \tilde{x}, P) \mid \mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]\}$

[6] $/n$ refers to the arity of the predicate.

[7] Processes and evaluation contexts appear as predicate arguments in this presentation to boost readability. Note that their syntax can be written as string literals to comply with the syntax of predicate logic.

Note that in UniversalCB$(A, \tilde{x}, P)$ and LocalCB$(A, B, \tilde{x}, P)$, $\tilde{x}$ is a single list of elements, rather than a sequence of elements passed as parameters to the predicates, and thus these predicates have fixed arities.

We define logical specification of traces both for finite and infinite cases according to the logical specification of configurations, and universal and local codebases, i.e., using $toFOL(\kappa, t)$, $\langle \mathcal{C}_\mathcal{U} \rangle$, and $\langle \mathcal{C}_\mathcal{L} \rangle$. Let $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \blacktriangleright \tau$. If τ is finite, i.e., $\tau = \kappa_0 \kappa_1 \cdots \kappa_n$ for some n, then its logical specification is defined as $toFOL(\tau) = \bigcup_{i=0}^{n} toFOL(\kappa_i, i) \bigcup \langle \mathcal{C}_\mathcal{U} \rangle \bigcup \langle \mathcal{C}_\mathcal{L} \rangle$. Otherwise, for infinite trace $\tau = \kappa_0 \kappa_1 \cdots$, $toFOL(\tau) = \bigcup_{\tau' \in prefix(\tau)} toFOL(\tau') \bigcup \langle \mathcal{C}_\mathcal{U} \rangle \bigcup \langle \mathcal{C}_\mathcal{L} \rangle$, where $toFOL(\tau') = \bigcup_{i=0}^{n} toFOL(\kappa_i, i)$, for $\tau' = \kappa_0 \kappa_1 \cdots \kappa_n$. It is straightforward to show that $toFOL(\tau)$ is injective and monotonically increasing.

3.2 A Class of Logging Specifications

We define the class of logging specifications $\mathcal{LS}_{call}$ that specify temporal relations among module invocations in concurrent systems. $\mathcal{LS}_{call}$ is the set of all logging specifications LS defined as $spec(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G is a set of Horn clauses, called *guidelines*, including clauses of the form

$$\forall t_0, \cdots, t_n, xs_0, \cdots, xs_n. \text{Call}(t_0, A_0, B_0, xs_0) \bigwedge_{i=1}^{n} \big(\text{Call}(t_i, A_i, B_i, xs_i) \wedge t_i < t_0\big) \wedge \\ \varphi(t_0, \cdots, t_n) \wedge \varphi'(xs_0, \cdots, xs_n) \implies \text{LoggedCall}(A_0, B_0, xs_0), \tag{2}$$

in which for all $j \in \{0, \cdots, n\}$, $A_j \in \mathcal{A}_\mathcal{U}$, $B_j \in \mathcal{A}_\mathcal{L}$, xs_j is a placeholder for a list of parameters passed to B_j, and $\text{Call}(t_j, A_j, B_j, xs_j)$ specifies the event of invoking module (subprocess) B_j by the top-level process A_j at time t_j with parameters xs_j. In (2), $\varphi(t_0, \cdots, t_n)$ is assumed to be a possibly empty conjunctive sequence of literals of the form $t_i < t_j$. Moreover, we define *triggers* and *logging events* as $Triggers(LS) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$ and $Logevent(LS) = (A_0, B_0)$, resp. *Logging preconditions* are predicates $\text{Call}(t_i, A_i, B_i, \tilde{x})$ for all $i \in \{1, \cdots, n\}$. As an additional condition, we assume that $Logevent(LS) \notin Triggers(LS)$.

Example 3.1 describes the logging specification for the breaking the glass policy specified in Section 1.2 for a medical records system, using (2).

Example 3.1 We revisit the example described in Section 1.2, a microservices-based medical records system, where breaking the glass entails logging the attempts to read patient medical history. Each microservice is treated as an agent, i.e., agents `Patient` and `Auth` correspond to Patient and Authorization microservices, resp. Let's assume that a user can break the glass by invoking `brkGlass` function from `Auth` agent. Moreover, reading patient medical history is accomplished by calling function `getMedHist` deployed by `Patient` agent. Indeed, these functions are treated as sub-agents in our calculus, i.e., we assume the existence of definitions:

- $\mathcal{C}_\mathcal{L}(\texttt{Auth})(\texttt{brkGlass}) = [\texttt{brkGlass}^{\texttt{Auth}}(u) \triangleq P]$ for some process P, and
- $\mathcal{C}_\mathcal{L}(\texttt{Patient})(\texttt{getMedHist}) = [\texttt{getMedHist}^{\texttt{Patient}}(p, u) \triangleq Q]$ for some process Q.

Then, the logging specification for the breaking-the-glass auditing policy is $LS = spec(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G includes the clause

$$\forall t_0, t_1, p, u. \text{Call}(t_0, \texttt{Patient}, \texttt{getMedHist}, [p, u]) \wedge \text{Call}(t_1, \texttt{Auth}, \texttt{brkGlass}, [u]) \wedge t_1 < t_0 \implies \\ \text{LoggedCall}(\texttt{Patient}, \texttt{getMedHist}, [p, u]).$$

In the clause above, t_0 and t_1 are timestamps where t_1 is preceding t_0. p refers to the patient identifier whose medical history is requested. u is the healthcare provider (user) identifier who breaks the glass and then attempts to read the medical history of patient p. Note that u is passed as an additional parameter to both `brkGlass` and `getMedHist`. In practice, this is accomplished in microservices using access tokens. An API gateway uses access tokens to communicate the identity of the service requester. One common approach to implement access tokens is by JSON Web Tokens standard [31].

3.3 Target System Model

We define the target system model, denoted by $\Pi_{\log}$, as an extension to Π with the following syntax and semantics. The instrumentation algorithm's job is to map a system specified in Π to a system in $\Pi_{\log}$.

3.3.1 Syntax

$\Pi_{\log}$ extends prefixes with $\alpha ::= \cdots \mid \mathsf{callEvent}(A, B, \tilde{x}) \mid \mathsf{emit}(A, B, \tilde{x}) \mid \mathsf{addPrecond}(x, A) \mid \mathsf{sendPrecond}(x, A)$. $\tilde{x}$ is considered as a single list of names in $\mathsf{callEvent}$ and emit, so that they have fixed arities.

$$\text{PI}\quad \frac{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, Q, \Delta, \Sigma, \Lambda)}$$

$$\text{CALL\_EV}\quad \frac{\Delta'(A) = \Delta(A) \cup \{\text{Call}(t, A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Delta'(A') = \Delta(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \qquad (t, \mathsf{callEvent}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta', \Sigma, \Lambda)}$$

$$\text{ADD\_PRECOND}\quad \frac{\Sigma'(A) = \Sigma(A) \cup \{x\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Sigma'(A') = \Sigma(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \qquad (t, \mathsf{addPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma', \Lambda)}$$

$$\text{SEND\_PRECOND}\quad \frac{y = \texttt{serialize}(\Delta(A))}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \qquad (t, \mathsf{sendPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, \bar{x}y.P, \Delta, \Sigma, \Lambda)}$$

$$\text{LOG}\quad \frac{\Sigma(A) \cup \Delta(A) \cup \Gamma_G \vdash \text{LoggedCall}(A, B, \tilde{x}) \qquad \Lambda'(A) = \Lambda(A) \cup \{\text{LoggedCall}(A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Lambda'(A') = \Lambda(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda')}$$

$$\text{NO\_LOG}\quad \frac{\Sigma(A) \cup \Delta(A) \cup \Gamma_G \nvdash \text{LoggedCall}(A, B, \tilde{x})}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda)}$$

Fig. 3. Unlabeled reduction semantics of Π_{log}.

A configuration κ, in Π_{log}, is defined as the quintuple $\kappa ::= (t, P, \Delta, \Sigma, \Lambda)$, with the following details. t is a timing counter. P is the process associated with the whole concurrent system. Processes in Π_{log} are defined similar to Π, without any extensions. $\Delta(\cdot)$ is a mapping that receives an agent identifier A and returns the set of logical preconditions (to log) that denote the events transpired locally in that agent. That is, $\Delta(A)$ is a set of predicates of the form $\text{Call}(t, A, B, \tilde{x})$. $\Sigma(\cdot)$ is a mapping that receives an agent identifier A and returns the set of all logical preconditions that have taken place in the triggers, i.e., in all agents $A' \in \mathcal{A}_{\mathcal{U}}$, where $(A', B) \in \mathit{Triggers}$ for some $B \in \mathcal{A}_{\mathcal{L}}$. That is, $\Sigma(A)$ is a set of predicates of the form $\text{Call}(t, A', B, \tilde{x})$, where $(A', B) \in \mathit{Triggers}$. These preconditions are supposed to be gathered by A from other agents A', in order to decide whether to log an event. $\Lambda(\cdot)$ is a mapping that receives an agent identifier A and returns the audit log recorded by that agent. $\Lambda(A)$ is a set of predicates of the form $\text{LoggedCall}(A, B, \tilde{x})$. The initial configuration is $\kappa_0 = (0, P_{\mathfrak{s}}, \Delta_0, \Sigma_0, \Lambda_0)$, where for any $A \in \mathcal{C}_{\mathcal{U}}$, $\Delta_0(A) = \Sigma_0(A) = \Lambda_0(A) = \emptyset$.

3.3.2 Semantics

We use judgment $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa \longrightarrow \kappa'$ to specify a step of reduction in Π_{log}. Figure 3 depicts the unlabeled reduction semantics of Π_{log}. $\mathcal{C}_{\mathcal{U}}$, $\mathcal{C}_{\mathcal{L}}$, and Γ_G may be elided in the specification of reduction steps since they are static and may be clear from the context.

Π_{log} inherits the reduction semantics of Π, according to rule PI. Rule CALL_EV gives the reduction with prefix $\mathsf{callEvent}(A, B, \tilde{x})$. In this case, Δ gets updated for agent A with information about the invocation of subprocess B^A. In rule ADD_PRECOND, reduction with the prefix $\mathsf{addPrecond}(x, A)$ is specified. In this case, x is added to Σ. Rule SEND_PRECOND is about the reduction with prefix $\mathsf{sendPrecond}(x, A)$. In this case, the set of logging preconditions that are collected by A, i.e., $\Delta(A)$, is converted to a transferable object (aka object serialization), e.g., a string of characters describing the content of $\Delta(A)$, and sent though link x. Let `serialize()` be the semantic function that handles this conversion. With prefix $\mathsf{emit}(A, B, \tilde{x})$, agent A is supposed to study whether the predicate $\text{LoggedCall}(A, B, \tilde{x})$ is logically derivable from the local set of preconditions, i.e., $\Delta(A)$, the set of preconditions that are collected by other agents involved in the enforcement of the logging specification, i.e., $\Sigma(A)$, and the set of guidelines Γ_G. If the predicate is derivable, then it is added to the audit log of A, i.e., $\Lambda(A)$. Otherwise, the log does not change. Rule LOG specifies the former case, whereas the rule NO_LOG specifies the latter.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1\cdots$, we use notation $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \blacktriangleright \tau$ to specify the generation of trace τ under the universal codebase $\mathcal{C}_{\mathcal{U}}$, local codebase $\mathcal{C}_{\mathcal{L}}$, and set of guidelines Γ_G, according to the reduction system, i.e., $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

The generated trace in Π_{log} out of a target system $\mathfrak{s}$, i.e., $\mathfrak{s} \Downarrow \tau$, can be defined in the same style as defined in Π, i.e., by some valid initial system in Π_{log} [8], the initial configuration κ_0 in Π_{log}, and the aforementioned reduction system for Π_{log}.

The residual log of a configuration is defined as $\mathit{logof}(\kappa) = \mathbb{L} = \bigcup_{A \in \mathcal{A}_{\mathcal{U}}} \Lambda(A)$, where $\kappa = (\_, \_, \_, \_, \Lambda)$. [9] This

[8] In Section 3.4 one such initial system is given by the instrumentation algorithm.

[9] Underscore is used as wildcard.

instantiates $\tau \rightsquigarrow \mathbb{L}$ for $\Pi_{\log}$ (Definition 2.4). Since $\mathbb{L}$ is a set of logical literals, it suffices to define $toFOL(\cdot)$ for audit logs as $toFOL(\mathbb{L}) = \mathbb{L}$, which completes the instantiation of $\lfloor \mathbb{L} \rfloor$ (Definition 2.11).

Note that arbitrary systems in $\Pi_{\log}$ do not guarantee any correctness of audit logging. However, there is a subset of systems in $\Pi_{\log}$ that provably satisfy this property. These systems use the extended prefixes (introduced as part of $\Pi_{\log}$ syntax) in a particular way for this purpose. In the following section, we introduce an instrumentation algorithm to map any system in Π to a system in $\Pi_{\log}$, and later prove that any instrumented system satisfies correctness results for audit logging.

3.4 Instrumentation Algorithm

Instrumentation algorithm $\mathcal{I}$ takes a Π system, defined in (1), and a logging specification $LS \in \mathcal{LS}_{call}$, defined in Section 3.2, and produces a system $\mathfrak{s}'$ in $\Pi_{\log}$ defined as $\mathfrak{s}' = \langle P'_{\mathfrak{s}}, \mathcal{C}'_{\mathcal{U}}, \mathcal{C}'_{\mathcal{L}} \rangle$, where $P'_{\mathfrak{s}} = (\nu \tilde{c})(\nu \tilde{c}')(A_1(\tilde{x}'_1) \mid A_2(\tilde{x}'_2) \mid \cdots \mid A_m(\tilde{x}'_m))$. $\tilde{c}'$ is the sequence of names of the form c_{ij} which are all fresh, i.e., they are not used already in (1). Moreover, it is assumed that sub-agent identifiers D_{ij} are also fresh, i.e., they are undefined in $\mathcal{C}_{\mathcal{L}}$ component of (1).

Intuitively, $\mathcal{I}$ works as follows.

(i) $\mathcal{I}$ adds new links c_{ij} between agents A_i and A_j, where A_i is the agent that includes a sub-agent whose invocation is considered a logging event, and A_j is some agent that includes a sub-agent whose invocation is a trigger for that logging event. c_{ij} is used as a link between A_i and A_j to communicate logging preconditions (by sendPrecond and addPrecond prefixes).

(ii) Regarding the invocation of a sub-agent B^A,
  (a) if the invocation of B^A is a trigger, then the execution of B^A must be preceded by callEvent prefix. This way, the invocation of B^A is stored in A's local set of logging precondition ($\Delta(A)$), according to the rule CALL_EV.
  (b) if the invocation of B^A is a logging event, then execution of B^A must be preceded by callEvent, similar to the case above. Next, it must communicate on appropriate links (c_{ij}s) with all other agents that are involved as triggers according to the logging specification. To this end, B^A is supposed to notify each of those agents to send their collected preconditions. After receiving all those preconditions from involved agents on the dedicated links, it adds them to $\Sigma(A)$. This is done using addPrecond prefixes, according to the rule ADD_PRECOND. Then, it studies whether the invocation must be logged, before following normal execution. This is facilitated by emit prefix (rules LOG and NO_LOG).
  (c) if the invocation of B^A is neither a trigger nor a logging event, then that sub-agent executes without any change in behavior.

(iii) Regarding the invocation of an agent A
  (a) if A includes a sub-agent B^A whose invocation is considered a trigger, then A must be able to receive and handle incoming requests for collected preconditions. This is done by adding a subprocess to A that always listens for requests on the dedicated link (c_{ij}) between itself and the agent that may send such requests. Upon receiving such a request, it sends back the preconditions, handled by prefix sendPrecond according to the rule SEND_PRECOND, and then continues to listen on the link.
  (b) if A does not include any trigger invocation of a sub-agent, then A executes without any changes.

Formally, the details of the returned system $\mathfrak{s}'$ are as follows:

(i) $\tilde{c}'$: is the sequence of all names c_{ij} where $(A_i, B) = Logevent(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and $(A_j, B') \in Triggers(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$.

(ii) $\mathcal{C}'_{\mathcal{L}}$:
  (a) $\mathcal{C}'_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq \mathsf{callEvent}(A, B, \tilde{x}).P]$, if $(A, B) \in Triggers(LS)$ and $\mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]$.
  (b) $\mathcal{C}'_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq \mathsf{callEvent}(A_0, B_0, \tilde{x}) \,.\, \bar{c_{01}} \,.\, \cdots \,.\, \bar{c_{0n}} \,.\, c_{01}(p_1) \,.\, \cdots \,.\, c_{0n}(p_n) \,.\, \mathsf{addPrecond}(p_1, A_0). \cdots \,.\, \mathsf{addPrecond}(p_n, A_0) \,.\, \mathsf{emit}(A_0, B_0, \tilde{x}).P]$, if $(A_0, B_0) = Logevent(LS)$, $\mathcal{C}_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq P]$, and $Triggers(LS) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$.
  (c) $\mathcal{C}'_{\mathcal{L}}(A)(B) = \mathcal{C}_{\mathcal{L}}(A)(B)$, otherwise.

(iii) $\mathcal{C}'_{\mathcal{U}}$:
  (a) If $(A_j, B) \in Triggers(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and A_i be the agent that $(A_i, B') = Logevent(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = [A_j(\tilde{x}, c_{ij}) \triangleq P | D_{ij}^{A_j}(c_{ij})]$, where $\mathcal{C}_{\mathcal{U}}(A_j) = [A_j(\tilde{x}) \triangleq P]$, and $\mathcal{C}'_{\mathcal{L}}(A_j)(D_{ij}) = [D_{ij}^{A_j}(c_{ij}) \triangleq c_{ij}.\mathsf{sendPrecond}(c_{ij}, A_j).D_{ij}^{A_j}(c_{ij})]$.
  (b) If $(A_j, B) \notin Triggers(LS)$ for any $B \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = \mathcal{C}_{\mathcal{U}}(A_j)$.

Note that D_{ij} is defined recursively to facilitate listening on c_{ij} indefinitely for incoming requests about

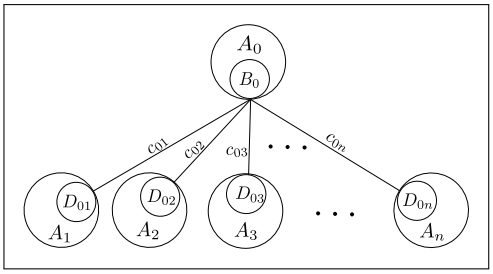

Fig. 4. Illustration of the established links in the instrumented concurrent system.

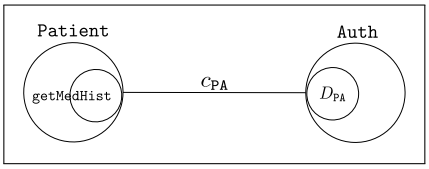

Fig. 5. Example: Illustration of the established link in the instrumented medical records system.

$$trim(\mathbf{0}) = \mathbf{0} \qquad trim(\alpha.P) = \begin{cases} trim(P) & \text{if } \alpha = c_{ij}, \bar{c_{ij}}, c_{ij}(x), \bar{c_{ij}}x, \\ & \mathsf{callEvent}(A, B, \tilde{x}), \mathsf{addPrecond}(x, A), \\ & \mathsf{sendPrecond}(x, A), \mathsf{emit}(A, B, \tilde{x}) \\ \alpha.trim(P) & \text{otherwise} \end{cases}$$

$$trim(P|Q) = \begin{cases} trim(P) & \text{if } Q = D^A_{ij} \text{ for some } i, j, A \\ trim(Q) & \text{if } P = D^A_{ij} \text{ for some } i, j, A \\ trim(P)|trim(Q) & \text{otherwise} \end{cases} \qquad trim((\nu x)P) = \begin{cases} trim(P) & \text{if } x = c_{ij} \text{ for some } i, j \\ (\nu x)trim(P) & \text{otherwise} \end{cases}$$

$$trim(C(\tilde{x})) = \begin{cases} trim(C(\tilde{y})) & \text{if } \tilde{x} = \tilde{y}, c_{ij} \text{ for some } i, j \\ C(\tilde{x}) & \text{otherwise} \end{cases}$$

Fig. 6. Function *trim*.

logging preconditions. In addition, since c_{ij} is fresh, $c_{ij} \notin fn(P)$. Therefore, P cannot communicate on this link, e.g., to compromise logging attempts. Figure 4 illustrates the established links between sub-agents of different agents according to the guideline defined in (2). These links are used to communicate logging preconditions between the logging event and the triggers.

Example 3.2 depicts how the medical records system described in Section 1.2 is instrumented according to the specified instrumentation algorithm, and the logging specification given in Example 3.1.

Example 3.2 In Example 3.1, $(\mathtt{Patient}, \mathtt{getMedHist})$ is the logging event, and $\{(\mathtt{Auth}, \mathtt{brkGlass})\}$ is the set of triggers. Applying the instrumentation algorithm changes the systems as follows. A new link $c_{\mathtt{PA}}$ is established between `Patient` and `Auth` agents. The definition of `brkGlass` is updated as $\mathcal{C}'_{\mathcal{L}}(\mathtt{Auth})(\mathtt{brkGlass}) = [\mathtt{brkGlass}^{\mathtt{Auth}}(u) \triangleq \mathsf{callEvent}(\mathtt{Auth}, \mathtt{brkGlass}, [u]).P]$, and the definition of `getMedHist` is updated as

$$\mathcal{C}'_{\mathcal{L}}(\mathtt{Patient})(\mathtt{getMedHist}) = [\mathtt{getMedHist}^{\mathtt{Patient}}(p, u) \triangleq \mathsf{callEvent}(\mathtt{Patient}, \mathtt{getMedHist}, [p, u]).\bar{c_{\mathtt{PA}}}.c_{\mathtt{PA}}(f).\mathsf{addPrecond}(f, \mathtt{Patient}).\mathsf{emit}(\mathtt{Patient}, \mathtt{getMedHist}, [p, u]).Q].$$

In addition subprocess $D_{\mathtt{PA}}$ is added to agent `Auth` that indefinitely responds to the requests from `Patient` on link $c_{\mathtt{PA}}$, defined as: $\mathcal{C}'_{\mathcal{L}}(\mathtt{Auth})(D_{\mathtt{PA}}) = [D^{\mathtt{Auth}}_{\mathtt{PA}}(c_{\mathtt{PA}}) \triangleq c_{\mathtt{PA}}.\mathsf{sendPrecond}(c_{\mathtt{PA}}, \mathtt{Auth}).D^{\mathtt{Auth}}_{\mathtt{PA}}(c_{\mathtt{PA}})]$. Figure 5 illustrates the established link between the sub-agents of the two agents.

3.4.1 Instantiation of $\approx$

According to Definition 2.5, semantics preservation relies on an abstraction of correspondence relation $\approx$ between source and target traces. In this section, we instantiate this relation for $\mathcal{I}$. We define the source and target trace correspondence relation as follows: $\tau_1\kappa_1 \approx \tau_2\kappa_2$ iff $\kappa_1 = P_1$, $\kappa_2 = (t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$, and $trim(P_2) = P_1$. Function *trim* is formally defined in Figure 6. Intuitively, it removes all prefixes, sub-agents, and link names that $\mathcal{I}$ may add to a process.

3.4.2 Main Results

Main properties include three results. The instrumentation algorithm $\mathcal{I}$ is semantics preserving, deadlock-free, correct. These are specified in Theorems 3.9, 3.10, and 3.15, resp.

Lemma 3.3 $P_{\mathfrak{s}} :\approx (0, P'_{\mathfrak{s}}, \Delta_0, \Sigma_0, \Lambda_0)$.

Proof. It is straightforward to show that $trim(P'_{\mathfrak{s}}) = P_{\mathfrak{s}}$, according to the definition of *trim* in Figure 6. □

Let $\longrightarrow^*$ be the reflexive and transitive closure of reduction relation $\longrightarrow$.

Lemma 3.4 *Let* $(t, \mathcal{E}[trim(P)], \Delta, \Sigma, \Lambda) \longrightarrow (t', \mathcal{E}[P'], \Delta', \Sigma', \Lambda')$. *Then, there exists some* t'', P'', Δ'', Σ'', *and* Λ'', *where*

- $(t, \mathcal{E}[P], \Delta, \Sigma, \Lambda) \longrightarrow^* (t'', \mathcal{E}[P''], \Delta'', \Sigma'', \Lambda'')$, *and*
- $trim(P') = trim(P'')$.

Proof. By induction on the structure of P. □

Lemma 3.5 *Let* $\tau_1 P_1 :\approx \tau_2(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$ *and* $P_1 \longrightarrow Q_1$. *Then, there exists some non-trivial trace* τ'_2 *such that* $(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2) \Downarrow \tau'_2$ *and* $\tau_1 P_1 Q_1 :\approx \tau_2 \tau'_2$.

Proof. By induction on the derivation of $P_1 \longrightarrow Q_1$, and application of Lemma 3.4. □

Lemma 3.6 *Let* $\tau_1 P_1 :\approx \tau_2(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$ *and* $P_1 \Downarrow \tau'_1$. *Then, there exists some trace* τ'_2 *such that* $(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2) \Downarrow \tau'_2$ *and* $\tau_1 \tau'_1 :\approx \tau_2 \tau'_2$.

Proof. By induction on the derivation of $P_1 \Downarrow \tau'_1$, and application of Lemma 3.5. □

Lemma 3.7 *Let* $\tau_1 P_1 :\approx \tau_2 \kappa_2$ *and* $\kappa_2 \longrightarrow \kappa'_2$. *Then, there exists some trace* τ'_1 *such that* $P_1 \Downarrow \tau'_1$ *and* $\tau_1 \tau'_1 :\approx \tau_2 \kappa_2 \kappa'_2$.

Proof. By induction on the derivation of $\kappa_2 \longrightarrow \kappa'_2$. □

Lemma 3.8 *Let* $\tau_1 P_1 :\approx \tau_2 \kappa_2$ *and* $\kappa_2 \Downarrow \tau'_2$. *Then, there exists some trace* τ'_1 *such that* $P_1 \Downarrow \tau'_1$ *and* $\tau_1 \tau'_1 :\approx \tau_2 \tau'_2$.

Proof. By induction on the derivation of $\kappa_2 \Downarrow \tau'_2$, and application of Lemma 3.7. □

Theorem 3.9 (Semantics preservation) $\mathcal{I}$ *is semantics preserving (Definition 2.5).*

Proof. Lemmas 3.3, 3.6, and 3.8 entail the result. In essence, Lemmas 3.3 and 3.6 satisfy the first condition in Definition 2.5, whereas Lemmas 3.3 and 3.8 satisfy the second condition in that definition. □

Theorem 3.10 (Deadlock-freeness) $\mathcal{I}$ *is deadlock-free (Definition 2.6).*

Proof. Let $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, and $\mathcal{I}(\mathfrak{s}, LS)$ be stuck following τ', i.e., for all configurations κ', we have $\mathcal{I}(\mathfrak{s}, LS) \not\Downarrow \tau'\kappa'$. According to Theorem 3.9, there exists some trace τ, such that $\mathfrak{s} \Downarrow \tau$ and $\tau :\approx \tau'$. Then, there are two possible cases:

(i) For any configuration κ, $\mathfrak{s} \not\Downarrow \tau\kappa$. Then, according to Definition 2.6, $\mathcal{I}(\mathfrak{s}, LS)$ is deadlock-free, vacuously.

(ii) There exists some configuration κ such that $\mathfrak{s} \Downarrow \tau\kappa$. Then, according to Lemma 3.5, there exists some non-trivial trace τ'' such that $tail(\tau') \Downarrow \tau''$ and $\tau\kappa :\approx \tau'\tau''$. The former entails that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'\tau''$, which contradicts with the assumption about $\mathcal{I}(\mathfrak{s}, LS)$ being stuck, and completes the proof.

□

Lemma 3.11 *The following propositions hold for closure in FOL and least Herbrand model (*$\mathcal{H}$*).*

(i) $Closure(\mathcal{H}(\Gamma)) = Closure(\Gamma)$

(ii) $\mathcal{H}(\Gamma) = \mathcal{H}(Closure(\Gamma))$

(iii) $Closure(Closure(\mathcal{H}(\Gamma) \cap \Gamma')) = Closure(\mathcal{H}(\Gamma) \cap \Gamma')$

Lemma 3.12 *Let* $(t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t', Q, \Delta', \Sigma', \Lambda')$. *Moreover, assume that* $\text{LoggedCall}(A, B, \tilde{x}) \in \Lambda'(A) - \Lambda(A)$. *Then, there exist some evaluation context* $\mathcal{E}$ *and some process* R *such that* $P = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$.

Proof. By induction on the derivation of unlabeled reduction relation, i.e., $(t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t', Q, \Delta', \Sigma', \Lambda')$. □

Lemma 3.13 *Let* $(t, P, \Delta, \Sigma, \Lambda) \Downarrow \tau(t', P', \Delta', \Sigma', \Lambda')$. *Moreover, assume that* $\text{LoggedCall}(A, B, \tilde{x}) \in \Lambda'(A) - \Lambda(A)$. *Then, there exist some evaluation context* $\mathcal{E}$ *and some process* R *along with trace* τ' *and configurations* κ_1 *and* κ_2 *such that* $\tau'\kappa_1\kappa_2 \in \mathit{prefix}(\tau(t', P', \Delta', \Sigma', \Lambda'))$, *and* $P_1 = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$, *where* $\kappa_1 = (\_, P_1, \_, \_, \_)$.

Proof. By induction on $(t, P, \Delta, \Sigma, \Lambda) \Downarrow \tau(t', P', \Delta', \Sigma', \Lambda')$ and application of Lemma 3.12. □

Let $[\cdots t]\tau$ denote the prefix of τ of length t.

Lemma 3.14 *If* $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, *and* $\tau' \rightsquigarrow \mathbb{L}$, *then* $\mathit{toFOL}(\mathbb{L}) = \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$.

Proof. We first show that $\mathit{toFOL}(\mathbb{L}) \subseteq \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. Let $\mathit{tail}(\tau') = (\_, \_, \_, \_, \Lambda)$. Assume that $\text{LoggedCall}(A, B, \tilde{x}) \in \mathit{toFOL}(\mathbb{L})$. Then, according to Lemma 3.13, there exist some trace $\hat{\tau}$ and configurations κ_1 and κ_2 such that $\hat{\tau}\kappa_1\kappa_2 \in \mathit{prefix}(\tau')$, $\kappa_1 = (\_, P_1, \_, \_, \_)$, and $P_1 = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$. By Theorem 3.9, we know that there exist some source trace τ_0 such that $\mathfrak{s} \Downarrow \tau_0$, and it simulates the target trace $\hat{\tau}\kappa_1\kappa_2$, i.e., $\tau_0 :\approx \hat{\tau}\kappa_1\kappa_2$. According to $\mathcal{I}$ definition, prefix emit can only appear in the log event of a logging specification. Therefore, preconditions of rule (2) are satisfied by $\Gamma_G \cup \mathit{toFOL}(\tau_0)$. Due to $\tau_0 \in \mathit{prefix}(\tau)$ and monotonicity of $\mathit{toFOL}()$, the precondition of rule (2) are satisfied by $\Gamma_G \cup \mathit{toFOL}(\tau)$. Therefore, $\text{LoggedCall}(A, B, \tilde{x}) \in \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$.

Next, we show the following: $\mathit{toFOL}(\mathbb{L}) \supseteq \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. Assume that $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. This entails $\text{Call}(t, A_0, B_0, \tilde{xs}) \in \mathit{toFOL}(\tau)$ for some t in which preconditions of rule (2) are satisfied. Then, $\mathit{tail}([\cdots t]\tau) = \mathcal{E}[B_0^{A_0}(\tilde{xs})]$ for some $\mathcal{E}$. According to Theorem 3.9, there exists some target trace $\hat{\tau}$ such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \hat{\tau}$, and $[\cdots t]\tau :\approx \hat{\tau}$. Since $\mathcal{I}$ is deadlock-free (Theorem 3.10), we know that $\mathcal{I}(\mathfrak{s}, LS)$ is not stuck after following execution trace $\hat{\tau}$. Let $\mathit{tail}(\hat{\tau}) = (\_, \hat{P}, \_, \_, \_)$. Then, $\mathit{trim}(\hat{P}) = \mathcal{E}[B_0^{A_0}(\tilde{xs})]$. We have

$$(\hat{t}, \mathcal{E}[\mathit{trim}(\hat{P})], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda}) \longrightarrow (\hat{t}+1, \mathcal{E}[\mathsf{callEvent}(A_0, B_0, \tilde{x}).\bar{c_{01}}.\cdots.\bar{c_{0n}}.c_{01}(p_1).\cdots.c_{0n}(p_n).\\ \mathsf{addPrecond}(p_1, A_0).\cdots.\mathsf{addPrecond}(p_n, A_0).\mathsf{emit}(A_0, B_0, \tilde{x}).R], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda})$$

By Lemma 3.4, we then have some trace τ_0, and configurations $\kappa_1, \cdots, \kappa_{3n+4}$ such that $(\hat{t}, \mathcal{E}[\hat{P}], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda}) \Downarrow \tau_0\kappa_0\kappa_1\cdots\kappa_{3n+4}$, where

- $\kappa_1 = (\_, \mathcal{E}'[B_0^{A_0}(\tilde{x})], \_, \_, \hat{\Lambda})$,
- $\kappa_2 = (\_, \mathcal{E}'[\mathsf{callEvent}(A_0, B_0, \tilde{x}).\bar{c_{01}}.\cdots.\bar{c_{0n}}.c_{01}(p_1).\cdots.c_{0n}(p_n).\mathsf{addPrecond}(p_1, A_0).\cdots.\mathsf{addPrecond}(p_n, A_0).\mathsf{emit}(A_0, B_0, \tilde{x}).R], \_, \_, \hat{\Lambda})$,
- ...
- $\kappa_{3n+3} = (\_, \mathcal{E}'[\mathsf{emit}(A_0, B_0, \tilde{x}).R], \_, \_, \hat{\Lambda})$, and
- $\kappa_{3n+4} = (\_, \mathcal{E}'[R], \_, \_, \hat{\Lambda}')$.

Note that $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \hat{\Lambda}'(A_0)$, which entails that if $\hat{\tau}\tau_0\kappa_1\cdots\kappa_{3n+4} \rightsquigarrow \hat{\mathbb{L}}$ then $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \hat{\mathbb{L}}$. Since $[\cdots(t+1)]\tau :\approx \hat{\tau}\tau_0\kappa_1\cdots\kappa_{3n+4}$ and $[\cdots(t+1)]\tau \in \mathit{prefix}(\tau)$, for any trace τ' where $\tau :\approx \tau'$, if $\tau' \rightsquigarrow \mathbb{L}$, then $\hat{\mathbb{L}} \subseteq \mathbb{L}$ due to monotonicity of log growth. This entails that $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathbb{L}$, and thus $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathit{toFOL}(\mathbb{L})$. □

Theorem 3.15 (Instrumentation correctness) *$\mathcal{I}$ is correct (Definition 2.7).*

Proof. Let $\mathfrak{s}$ be a source system and LS be a logging specification defined as Section 3.2. If $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then we need to show that $\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) = LS(\tau)$. By Lemma 3.14, we have $\mathit{toFOL}(\mathbb{L}) = \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. By Lemma 3.11, we have

$$\begin{aligned} LS(\tau) &= \mathit{Closure}(\mathit{Closure}(\mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau))) \cap \mathit{FOL}(\{\text{LoggedCall}\})) \\ &= \mathit{Closure}(\mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})) \end{aligned}$$

This entails the result. □

4 Related Work

Majority of previous work on audit logging in concurrent environments focus on audit log analysis (e.g., [32]) and security concerns regarding in transit and/or at rest log information (e.g., [33,34,35]). Studies regarding

the collection of logs from multiple monitors in distributed intrusion detection systems are such instances (e.g., [36,37]). However, previous work do not reflect on the generation of the log, and assume that the audit log is given. This line of work includes studies on the security of audit logs in terms of their secrecy and integrity within concurrent environments. For example, Yavuz et al. [34] propose a logging scheme that guarantees forward security, employing cryptographic techniques. In the same line of work, Böck et al. [38] propose a system that ensures that logs are trustworthy. Our work is however orthogonal to the notion of audit log security. Using cryptographic techniques to ensure verifiable confidentiality and integrity of the audit log does not guarantee it to be correct. We employ a semantic framework by which the content of the audit log can be judged against the execution trace of the concurrent system given in π calculus (Definition 2.3).

Cederquist et al. [39] and Corin et al. [40] propose predicate logic frameworks to specify and enforce accountability requirements in distributed systems. The former proposes a framework that ensures user accountability in discretionary access control. The latter studies user accountability in access to personal data that are associated with usage policies defined by the owner of data, and can be distributed among users. Jagadeesan et al. [41] use turn-based games to analyze distributed accountability systems. Guts. et al. [42] use static type enforcement to assure that a distributed system generates sufficient audit logs. However, our approach is dynamic and relies on instrumentation techniques that can be applied to legacy systems which may inherently suffer from the lack of correct audit logging mechanisms. With respect to system instrumentation for auditing purposes, our work is related to the language proposed by Martin et al. [43] that facilitates querying runtime behavior of a program.

Another line of work employs logs to record proof of legitimate access to system resources. Vaughn et al. [44] propose an architecture based on trusted kernels that rely on such logged proofs. Another related work is the a posteriori compliance control system [45] that verifies legitimacy of access after the fact, using a trust-based logical framework that focuses on a limited set of operations. However, our logical framework is used to specify invocation of any arbitrary operation as a precondition to log, or the logging event.

Audit logs can be considered a form of provenance [46]. CamFlow [47] is an auditing and provenance capture utility in Linux that can easily integrate with distributed systems. Pasquier et al. [48] make strong case for accountability, data provenance and audit in the IoT. AccessProv [5] is proposed as an instrumentation tool that rewrites legacy Java applications for provenance and finds bugs in authorization systems. Kacianka et al. [49] propose a formal model of accountability for cyber-physical systems.

Amir-Mohammadian et al. [11] propose a semantic framework for audit logging based on the theory of information algebra [50,12]. Their implementation model is restricted to sequential computation. This model is therefore insufficient to apply on concurrent systems where logging preconditions and logging events may transpire in different execution threads. Moreover, their implementation model is restricted to deterministic system behavior. Our work generalizes the application of information-algebraic semantics of audit logging to concurrent environments, which naturally behave non-deterministically at runtime. We show that the semantic framework is inclusive enough for this purpose. Similar to [11], we propose a provably correct instrumentation algorithm. However, our algorithm retrofits a concurrent system (rather than a simple sequential program) according to a formal description of audit logging requirements. Information-algebraic semantic framework for audit logging has also been used to enhance dynamic integrity taint analysis through after-the-fact study of audit logs [8,9]. This line of work introduces maybe-tainted tags for data objects and proposes an implementation model on a core functional object-oriented calculus that provably ensures correctness of generated audit logs. However, it does not address the problem of deploying audit logging in concurrent environments.

Recently, Justification Logic [51] is used to formally characterize auditing of computational units [52,53,54] which result in programming languages that enable applications to study their own audit trails and decide accordingly. This is a separate theoretical problem than what we are aiming in this work.

Microservices-based approach [55,56] to software deployment is an application of our implementation model, that we aim to study in future in a greater detail. Accountability plays a significant role as part of the access control framework in microservices-based systems [57], including platform-specific monitoring techniques, e.g., in Azure Kubernetes Service [58]. Smith et al. [59] have proposed a provenance management system, including provenance logger, for microservices-based applications. Camilli et al. [60] have proposed a semantics for microservices based on Petri nets. Our approach is, however, language-based and relies on process calculi. Jolie [20] is the major programming language for the deployment of microservices, whose semantics [21,22] is defined as a process calculus, heavily influenced by π-calculus.

5 Future Work and Conclusion

In this paper, we have proposed an implementation model to enforce correct audit logging in concurrent environments. In essence, we have proposed an algorithm that instruments legacy concurrent systems according to a formal specification of audit logging requirements. We use Horn clause logic to specify these logging requirements, which assert temporal relations among the events that transpire in different concurrent components of

the system. We have proven that our algorithm is semantics preserving, i.e., the instrumented system behaves similar to the original system, modulo operations that correspond to audit logging. Moreover, we have proven that our algorithm guarantees correct audit logs. This ensures that the instrumented system avoids missing any logging event, as well as logging unnecessary events. Correctness of audit logs are defined according to an information-algebraic semantic framework. In this semantic framework, information containment is used to compare the runtime behavior vs. the generated audit log.

We have argued that the our instrumentation algorithm proposes a model to implement audit logging in real concurrent systems, e.g., in microservices-based medical records systems (Examples 3.1 and 3.2). In this paper, we have aimed at the formal specification of the model on an abstract core calculus to demonstrate the main ideas for future deployment. In future work, we intend to consider real-world language settings, relying on the fundamental results established in this work. In particular, we are aiming to deploy our existing instrumentation algorithm in Spring Boot [18], a Java microservices framework. Indeed, to provide formal guarantees of audit logging correctness in such real-world settings that are implemented using our model, translation of systems to Π systems is required.

Another area of interest is to extend the class of logging specifications, and hence implementation models. Our current class focuses on function invocations within each agent of the system, and it is limited to Horn clauses. While a great percentage of system events can be specified in this class, we need other classes of logging specifications for certain purposes. For example, consider the effect of revoking break-the-glass status for a user on the specification of audit logging requirements. Moreover, auditing usually includes the log of message transmissions between specific agents, which is not supported by what we have introduced in this paper.

Since our model of concurrency is based on a process calculi, message-passing is used for IPC. This necessitates the exploration of models that study the specification and enforcement of correct audit logging in concurrent environments which handle IPC through alternative approaches, e.g., shared memory and/or files.

References

[1] C. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," *CoRR*, vol. abs/1906.11507, 2019.

[2] H. DeYoung, D. Garg, L. Jia, D. K. Kaynar, and A. Datta, "Experiences in the logical specification of the HIPAA and GLBA privacy laws," in *WPES 2010*, 2010, pp. 73–82.

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

[4] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How do system administrators resolve access-denied issues in the real world?" in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 348–361.

[5] F. Capobianco, C. Skalka, and T. Jaeger, "ACCESSPROV: Tracking the provenance of access control decisions," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.

[6] W. Zhang, Y. Chen, T. Cybulski, D. Fabbri, C. Gunter, P. Lawlor, D. Liebovitz, and B. 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*. ACM, 2014, pp. 1182–1192.

[7] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Foundations for auditing assurance," in *Layered Assurance Workshop*, 2015.

[8] S. Amir-Mohammadian and C. Skalka, "In-depth enforcement of dynamic integrity taint analysis," in *Programming Languages and Analysis for Security*, 2016.

[9] C. Skalka, S. Amir-Mohammadian, and S. Clark, "Maybe tainted data: Theory and a case study," *Journal of Computer Security*, vol. 28, no. 3, pp. 295–335, April 2020.

[10] "Top 10-2017 A10-Insufficient Logging & Monitoring," https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A10-Insufficient_Logging%252526Monitoring, 2017, accessed: 2020-06-04.

[11] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Correct audit logging: Theory and practice," in *Principals of Security and Trust*, 2016, pp. 139–162.

[12] J. Kohlas and J. Schmid, "An algebraic theory of information: An introduction and survey," *Information*, vol. 5, no. 2, pp. 219–254, 2014.

[13] P. Matthews and H. Gaebel, "Break the glass," in *HIE Topic Series*. Healthcare Information and Management Systems Society, 2009.

[14] "Learning the lessons of the Dixons Carphone breach," https://www.zdnet.com/article/learning-the-lessons-of-the-dixons-carphone-breach/, 2020, accessed: 2020-01-15.

[15] "SWI Prolog," https://www.swi-prolog.org/, accessed: 2019-09-15.

[16] "XSB Prolog," https://xsb.com/xsb-prolog, accessed: 2019-09-15.

[17] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[18] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, and S. Deleuze, "Spring boot reference guide," *Part IV. Spring Boot features*, vol. 24, 2013.

[19] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA 2005*, 2005, pp. 345–364.

[20] "Jolie Programming Language," https://www.jolie-lang.org/, accessed: 2019-09-18.

[21] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "Sock: a calculus for service oriented computing," in *International Conference on Service-Oriented Computing*. Springer, 2006, pp. 327–338.

[22] F. Montesi and M. Carbone, "Programming services with correlation sets," in *International Conference on Service-Oriented Computing*. Springer, 2011, pp. 125–141.

[23] "Microservices in Healthcare: Granulate to Accelerate," https://vicert.com/local/resources/assets/pdf/WhitePaper_Microservices%20in%20Healthcare.pdf, 2019, accessed: 2020-02-25.

[24] "Global Microservices In Healthcare Market Will Reach USD 519 Million By 2025," https://www.globenewswire.com/news-release/2019/03/14/1753060/0/en/Global-Microservices-In-Healthcare-Market-Will-Reach-USD-519-Million-By-2025-Zion-Market-Research.html, 2019, accessed: 2019-09-01.

[25] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[26] Ú. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, Cornell University, 2003.

[27] S. Amir-Mohammadian, "A formal approach to combining prospective and retrospective security," Ph.D. dissertation, The University of Vermont, July 2017.

[28] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[29] R. Milner, *Communication and concurrency*. Prentice hall New York etc., 1989, vol. 84.

[30] J. Parrow, "An introduction to the π-calculus," in *Handbook of Process Algebra*. Elsevier, 2001, pp. 479–543.

[31] "Introduction to JSON Web Tokens," https://jwt.io/introduction/, 2019, accessed: 2019-09-02.

[32] A. Mounji, B. L. Charlier, D. Zampuniéris, and N. Habra, "Distributed audit trail analysis," in *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*, 1995, pp. 102–113.

[33] A. J. Lee, P. Tabriz, and N. Borisov, "A privacy-preserving interdomain audit framework," in *Proceedings of the 2006 ACM Workshop on Privacy in the Electronic Society, WPES 2006, Alexandria, VA, USA, October 30, 2006*, 2006, pp. 99–108.

[34] A. A. Yavuz and P. Ning, "BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems," in *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, 2009, pp. 219–228.

[35] R. Accorsi, "Bbox: A distributed secure log architecture," in *Public Key Infrastructures, Services and Applications - 7th European Workshop, EuroPKI 2010, Athens, Greece, September 23-24, 2010. Revised Selected Papers*, 2010, pp. 109–124.

[36] Y. Wu, B. Foo, Y. Mei, and S. Bagchi, "Collaborative intrusion detection system (CIDS): A framework for accurate and efficient IDS," in *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, 2003, pp. 234–244.

[37] V. Yegneswaran, P. Barford, and S. Jha, "Global intrusion detection in the DOMINO overlay system," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, 2004.

[38] B. Böck, D. Huemer, and A. M. Tjoa, "Towards more trustable log files for digital forensics by means of "trusted computing"," in *AINA 2010*. IEEE Computer Society, 2010, pp. 1020–1027.

[39] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini, "Audit-based compliance control," *International Journal of Information Security*, vol. 6, no. 2-3, pp. 133–151, 2007.

[40] R. Corin, S. Etalle, J. I. den Hartog, G. Lenzini, and I. Staicu, "A logic for auditing accountability in decentralized systems," in *FAST 2004*, 2004, pp. 187–201.

[41] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a theory of accountability and audit," in *ESORICS 2009*, 2009, pp. 152–167.

[42] N. Guts, C. Fournet, and F. Z. Nardelli, "Reliable evidence: Auditability by typing," in *European Symposium on Research in Computer Security*. Springer, 2009, pp. 168–183.

[43] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *OOPSLA 2005*. ACM, 2005, pp. 365–383.

[44] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic, "Evidence-based audit," in *CSF 2008*, 2008, pp. 177–191.

[45] S. Etalle and W. H. Winsborough, "A posteriori compliance control," in *SACMAT 2007*, 2007, pp. 11–20.

[46] W. Ricciotti, "A core calculus for provenance inspection," in *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2017, pp. 187–198.

[47] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 405–418.

[48] T. Pasquier, J. Singh, J. Powles, D. Eyers, M. Seltzer, and J. Bacon, "Data provenance to audit compliance with privacy policy in the internet of things," *Personal and Ubiquitous Computing*, vol. 22, no. 2, pp. 333–344, 2018.

[49] S. Kacianka and A. Pretschner, "Understanding and formalizing accountability for cyber-physical systems," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2018, pp. 3165–3170.

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

[51] S. Artemov, "Justification logic," in *European Workshop on Logics in Artificial Intelligence*. Springer, 2008, pp. 1–4.

[52] F. Bavera and E. Bonelli, "Justification logic and audited computation," *Journal of Logic and Computation*, vol. 28, no. 5, pp. 909–934, 2015.

[53] W. Ricciotti and J. Cheney, "Strongly normalizing audited computation," *arXiv preprint arXiv:1706.03711*, 2017.

[54] W. Ricciotti and J. Cheney, "Explicit auditing," *arXiv preprint arXiv:1808.00486*, 2018.

[55] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[56] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, "Microservices: a language-based approach," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 217–225.

[57] M. McLarty, R. Wilson, and S. Morrison, *Securing Microservice APIs*. O'Reilly Media, Inc., 2018.

[58] M. Wasson, "Monitoring a microservices architecture in Azure Kubernetes Service (AKS)," https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring, 2019, accessed: 2019-09-15.

[59] W. Smith, T. Moyer, and C. Munson, "Curator: provenance management for modern distributed systems," in *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[60] M. Camilli, C. Bellettini, L. Capra, and M. Monga, "A formal framework for specifying and verifying microservices based process flows," in *International Conference on Software Engineering and Formal Methods*. Springer, 2017, pp. 187–202.