An Implementation Model for Correct Audit Logging in Cyber-Physical Systems

Sepehr Amir-Mohammadian and Afsoon Yousefi Zowj

Dept. of Computer Science, University of the Pacific, Stockton, CA
{samirmohammadian,ayousefizowj}@pacific.edu

Abstract. The widespread presence of cyber-physical systems necessitates a reliable assurance mechanism for audit logging across various discrete and continuous components of these systems. This paper explores an implementation model for cyber-physical systems. We introduce an algorithm designed to equip such systems in accordance with a formal specification of audit logging requirements, which provably ensures the generation of accurate audit logs in any instrumented system. The accuracy of the audit log is studied within an information-algebraic semantic framework of audit logging.

Keywords: Audit Logs · Cyber-Physical Systems · Programming Languages · Security

1 Introduction

Audit logs play a crucial role in enhancing security across various domains by providing a detailed record of system activities and events. They serve as an indispensable source of information for detecting and investigating security incidents, breaches, and unauthorized access attempts [9]. By capturing essential details such as user actions, system configurations, network traffic, and application activities, audit logs enable security teams to monitor, analyze, and respond to potential threats effectively. Moreover, audit logs serve as a deterrent to malicious actors, as the knowledge of being monitored can dissuade unauthorized activities.

In cyber-physical systems (CPSs), where the integration of physical processes with computing and networking capabilities is prevalent, audit logging becomes even more critical. These systems are subject to various security threats, including cyber attacks targeting critical infrastructure [22], industrial control systems [2] and autonomous vehicles (AVs) [16]. Audit logging in CPSs enables the tracking of interactions between physical and digital components, which provides insights into system behavior, anomalies, and potential vulnerabilities. For instance, in smart grid systems, audit logs can help detect and prevent unauthorized access to energy distribution networks, ensuring the reliability and integrity of the grid [18]. Similarly, in AVs, audit logs are essential for recording sensor data, decision-making processes, and helping with the forensic analysis and liability attribution in case of accidents or cyber attacks [10].

In recent years, information-algebraic models [17], have emerged as valuable semantic frameworks for audit logging. These models interpret audit logs and the runtime structure of processes as information-algebraic elements, providing intuitive insights into their information content. The reliability of audit logging hinges on this interpretation, which compares the information content of audit logs with that of the program at runtime. Leveraging this framework, an implementation model has been proposed for linear [5] and concurrent [6] computations, ensuring accurate audit logging through instrumentation techniques.

In this paper, we explore the application of the aforementioned information-algebraic framework in CPSs. To this end, we use a programming-linguistic model of CPSs, known as hybrid programs (HPs) [24–26]. Hybridity in the context of HPs refers to the simultaneous presence of discrete computational elements and continuous physical dynamics inherent in CPSs. HPs serve as a fundamental programming language for CPSs, allowing for the specification of integrated discrete and continuous behaviors. We use a variant of HPs, where the semantics is specified operationally (rather than denotationally) to facilitate the specification of audit log generation at runtime. Our formalism provides a model for developing CPS tools with correctness guarantees. This language model offers the following features: i) The language model supports the hybrid nature of CPSs, accommodating both discrete computational steps and continuous physical behavior. ii) Leveraging a variant of HPs, our approach benefits from concise syntax and semantics. This facilitates the description of a broad spectrum of hybrid-dynamic systems. iii) In order to articulate auditing requirements effectively, our model incorporates timestamps as part of the runtime environment. This enables the specification of the ordering of significant events. iv) Fundamental to specifying auditing requirements is the abstraction of secure operations. Named functions serve as these fundamental units. They offer a versatile tool for expressing auditing needs across different languages and systems.

Utilizing the formalism with the aforementioned features empowers us to model CPSs that ensure the accurate generation of audit logs in line with the developed semantic framework. In this paper, we present an instrumentation algorithm designed to modify an HP based on precise audit logging requirements. We demonstrate the correctness of this algorithm, as per the semantic framework, which ensures that the instrumented concurrent system produces accurate audit logs. The implementation of audit logging policies through code instrumentation offers a separation of policy from code, which lays the groundwork for studying the effectiveness of enforcement mechanisms using formal methods. Additionally, it can be automatically applied to legacy code to bolster system accountability.

An Illustrative Example Let's consider a simplified scenario to illustrate the necessity of studying correct audit logging in CPSs. We will revisit this example multiple times throughout the paper. Our example is an AV system that includes a controller and a physical machinery, and the task is to log a significant event like *hard braking*. Here is how such a sequence of events might unfold, culminating in a log entry: i) The controller continuously monitors the vehicle's speed and uses sensors to detect objects in its path. For instance, as the vehicle cruises at 60 mph, a pedestrian unexpectedly steps onto the road. ii) The detection of a pedestrian in the vehicle's path triggers the controller's obstacle avoidance algorithm. The controller assesses the distance and speed relative to the obstacle. It calculates that normal braking will not suffice to avoid a collision. iii) Based on the controller's decision, a command is sent to the physical machinery to execute a hard brake. iv) The physical machinery applies the vehicle's brakes forcefully and quickly to reduce speed dramatically in a short time. As the hard braking occurs, this critical action is captured as a log event. The system records this event along with pertinent details such as the vehicle's speed at the time of braking, the time, and the location. Retroactively, the log can be reviewed for compliance with safety protocols or used for improving the decision-making algorithm based on real-world outcomes. In this scenario, the sequence of detection, decision, execution, and logging is crucial for ensuring both the immediate safety of the vehicle's occupants and others on the road, as well as for long-term improvements and accountability in AV operations.

Paper Outline The rest of the paper is organized as follows: In Section 2, we review the information-algebraic semantic framework for correct audit logging. In Section 3, we explore the implementation model for HPs by specifying the source and target language models, as well as the instrumentation algorithm that maps an HP from the source language to an HP in the target language. In addition, towards the end of this section the main results of the work are specified. Section 4 discusses the related work. Finally, Section 5 concludes the paper.

2 Semantics of Audit Logging

In this section, we explore the information-algebraic semantics of audit logging in an informal manner. The content of this section has originally been explored elsewhere formally [5]. We avoid delving into the detailed formal presentation here for the sake of economy of space.

2.1 Information-Algebraic Semantic Framework

Audit logs abstract program states into configurations κ, transitioning via $\kappa \longrightarrow \kappa'$. A program trace τ is a sequence of these configurations, with $\mathcal{T}$ being all such sequences and $\mathit{prefix}(\tau)$ their initial segments. A program $\mathfrak{p}$ or configuration κ can produce a trace τ', and if $\tau \in \mathit{prefix}(\tau')$, we denote this as $\mathfrak{p} \Downarrow \tau$ or $\kappa \Downarrow \tau$.

Information algebra is used to define audit log correctness. In Section 2.2, we use this framework to represent specific audit logging criteria. An information algebra (Φ, Ψ) consists of Φ, a semigroup of information elements, and Ψ, a lattice of query domains. It includes combination ($\otimes$) and focusing ($^{\Rightarrow}$) operations that adhere to specific properties governing their interactions. [17]. In the information algebra framework, elements $X, Y \in \Phi$ are combined using $X \otimes Y$, and information X is selectively extracted using $X^{\Rightarrow E}$ based on the querying domain $E \in \Psi$. The combination of information elements introduces a partial order, $\preccurlyeq$, where $X \preccurlyeq Y$ means that combining X and Y results in Y, indicating Y contains all the information of X.

In audit logging, execution traces are mapped as information elements through $\lfloor \cdot \rfloor : \mathcal{T} \to \Phi$, which is injective and monotonically increasing, indicating that longer traces contain more information than their prefixes.

Audit logging requirements are defined abstractly as a *logging specification*, adaptable to different execution models and information formats. Concrete implementations of this specification are detailed in later sections to guide audit logging practices. The logging specification $LS : \mathcal{T} \to \Phi$ describes the information that should be logged for a given execution trace τ. Note that $\lfloor \cdot \rfloor$ and LS both map traces to information elements, but serve different purposes: $\lfloor \tau \rfloor$ captures all information in a trace, while $LS(\tau)$ specifies what should be logged during that trace's execution.

An audit log, denoted as $\mathbb{L}$, collects data during runtime, with $\mathcal{L}$ representing the set of all possible logs. To assess a log's correctness, we compare its content to the trace that produced it, using a mapping $\lfloor \cdot \rfloor : \mathcal{L} \to \Phi$. This mapping must satisfy that larger audit logs contain more information, by ensuring that $\lfloor \mathbb{L} \rfloor \preccurlyeq \lfloor \mathbb{L}' \rfloor$ for $\mathbb{L} \subseteq \mathbb{L}'$. An audit log $\mathbb{L}$ is deemed *correct* with respect to a logging specification LS and a program trace τ if both $\lfloor \mathbb{L} \rfloor \preccurlyeq LS(\tau)$ and $LS(\tau) \preccurlyeq \lfloor \mathbb{L} \rfloor$ are satisfied. The former indicates the necessity of the information in the audit log, while the latter signifies the sufficiency of that information.

In systems generating audit logs at runtime, these logs are integral to the system's configuration. We define $\mathit{logof}(\kappa)$ to extract audit logs from any given program configuration, κ. This log expands as execution progresses. *logof* is used to represent the logs accumulated over a trace. The residual log of a finite program trace τ is $\mathbb{L}$, denoted by $\tau \rightsquigarrow \mathbb{L}$, iff $\tau = \kappa_0 \kappa_1 \cdots \kappa_n$ and $\mathit{logof}(\kappa_n) = \mathbb{L}$.

An instrumentation algorithm $\mathcal{I}(\mathfrak{p}, LS)$ modifies a program $\mathfrak{p}$ based on a logging specification LS to add required audit logging. The process results in a target program that generates specified logs.

The instrumentation algorithm should maintain the original program's semantics while adding audit logging, ensuring the target program behaves similarly to the source, except for logging differences. This concept, called *semantics preservation*, uses a *correspondence relation* $:\approx$ to link source and target traces, adaptable to various program implementations. The instrumentation algorithm $\mathcal{I}$ is semantics-preserving if it ensures that for every trace τ of the original program $\mathfrak{p}$, there is a corresponding trace τ' in the instrumented program such that $\tau :\approx \tau'$, and vice versa. This applies to all programs $\mathfrak{p}$ and logging specifications LS where $\mathcal{I}(\mathfrak{p}, LS)$ is applicable.

Intuitively, $\mathcal{I}$ is correct if the instrumented program generates audit logs that are correct with respect to the logging specification and the source trace. This criterion must hold for any source program, any logging specification, and any possible log generated by the instrumented program.

2.2 Instantiation of Logging Specification

The definition of a logging specification maps program traces to information elements, using information algebra for practical application. We focus on logical specifications for audit logs, favoring first-order logic (FOL) for its expressive power and compatibility with logic programming engines. Other variants of logical frameworks can also be used for this purpose.

To instantiate information algebra, we need to instantiate the set Φ of information elements, the lattice Ψ of querying domains, and the combination and focusing operators. To this end, we consider Φ_{FOL} as the set of closed sets of FOL formulas, established under a proof-theoretic deductive system. We instantiate Ψ_{FOL} by considering a query domain as a subset of FOL, defined over specific predicate symbols, denoted by $FOL(S)$, where $S \in \mathit{Preds}$ is a subset of predicate symbols. Finally, combination involves taking the closure of the union of two sets of formulas, and focusing entails taking the closure of the intersection of an information element and a query domain.

To utilize (Φ_{FOL}, Ψ_{FOL}) as a framework for audit logging, we also need to instantiate the mapping $\lfloor \cdot \rfloor$. This mapping interprets both execution traces and audit logs as information elements. We define $\mathit{toFOL}(\cdot) : (\mathcal{T} \cup \mathcal{L}) \rightarrow FOL(\mathit{Preds})$ as an injective and monotonically increasing function. Then, to interpret both traces and logs as information elements in (Φ_{FOL}, Ψ_{FOL}), we instantiate $\lfloor \cdot \rfloor = \mathit{Closure}(\mathit{toFOL}(\cdot))$.

We define a logging specification LS using a set of FOL rules Γ and a set of specific predicate symbols S. This specification maps a trace τ to predicates in S derivable from Γ and events in τ, i.e., a logging specification $\mathit{spec}(\Gamma, S) : \mathcal{T} \rightarrow \Phi_{FOL}$ is defined as $\mathit{spec}(\Gamma, S) = \tau \mapsto (\lfloor \tau \rfloor \otimes \mathit{Closure}(\Gamma))^{\Rightarrow FOL(S)}$.

3 Implementation Model on Cyber-Physical Systems

This section introduces an implementation model aimed at ensuring accurate audit logging in CPSs. We employ HPs to define the CPS and put forth an instrumentation algorithm that modifies the program according to a given logging specification. Furthermore, we detail and establish pertinent properties, notably including the correctness of the instrumentation algorithm.

In Section 3.1, we discuss the syntax and semantics of the source program model. Section 3.2 introduces a class of logging specifications adept at specifying temporal relations among computational events in hybrid-dynamic systems. Following this, Section 3.3 outlines the syntax and semantics of hybrid programs enriched with audit logging capabilities. Finally, in Section 3.4, we delve into the instrumentation algorithm and elucidate the properties it adheres.

3.1 Source HP Model

We consider HPs as our source language model, refered by $\mathcal{HP}$. Syntax and semantics of HPs [26] are defined in the following, which are grounded in real arithmetic polynomial terms and FOL of real arithmetic.

O1

$$(t, \omega, x := e) \longrightarrow (t+1, \omega[x \mapsto \omega\llbracket e \rrbracket], \epsilon)$$

O2

$$\frac{r \in \mathbb{R}}{(t, \omega, x := *) \longrightarrow (t+1, \omega[x \mapsto r], \epsilon)}$$

O3

$$\frac{\omega \vDash P}{(t, \omega, ?P) \longrightarrow (t+1, \omega, \epsilon)}$$

O4

$$\frac{\mathcal{C}(f) = [f(\bar{x}) = \alpha] \qquad \omega\llbracket \bar{e} \rrbracket = \bar{r}}{(t, \omega, f(\bar{e})) \longrightarrow (t+1, \omega[\bar{x} \mapsto \bar{r}], \alpha)}$$

O5

$$\frac{\exists r \geq 0, \varphi : [0, r] \to \mathcal{S}. \qquad \big((\varphi \text{ solves } x' = e \text{ on } [0, r]) \wedge (\forall t \in [0, r]. \varphi(t) \vDash P)\big)}{(t, \varphi(0), x' = e \ \&\ P) \longrightarrow (t + r, \varphi(r), \epsilon)}$$

O6

$$\frac{i = 1, 2}{(t, \omega, \alpha_1 \cup \alpha_2) \longrightarrow (t+1, \omega, \alpha_i)}$$

O7

$$\frac{(t, \omega, \alpha) \longrightarrow (t', \omega', \alpha')}{(t, \omega, \mathcal{E}[\alpha]) \longrightarrow (t', \omega', \mathcal{E}[\alpha'])}$$

O8

$$\frac{\alpha_1 \equiv \alpha_1' \qquad \alpha_2 \equiv \alpha_2' \qquad (t, \omega, \alpha_1) \longrightarrow (t', \omega', \alpha_2)}{(t, \omega, \alpha_1') \longrightarrow (t', \omega', \alpha_2')}$$

Fig. 1: Semantics of HPs.

We define polynomial terms e with rational coefficients over a countably infinite set of variables $\mathcal{V}$. A polynomial term e takes the form $e ::= x \mid c \mid e.e \mid e + e$, where $x \in \mathcal{V}$ and $c \in \mathbb{Q}$ (a rational number). A state, ω, is a mapping from variables to real numbers, i.e., $\omega : \mathcal{V} \to \mathbb{R}$. We let $r \in \mathbb{R}$ to range over real numbers throughout the paper. The semantics of a polynomial term e is determined by a state, $\omega\llbracket e \rrbracket$, given in the standard form. For example, $\omega\llbracket x \rrbracket = \omega(x)$, and $\omega\llbracket e.e' \rrbracket = \omega\llbracket e \rrbracket.\omega\llbracket e' \rrbracket$.

The FOL of real arithmetic is syntactically defined by $P ::= e = e \mid e \geq e \mid \neg P \mid P \wedge P \mid \exists x P$. Additional syntactic structures, such as disjunction, implication, universal quantification, etc., can be defined using this minimal syntax. A state ω models a predicate P, denoted by $\omega \vDash P$ in the standard format. For example, i) $\omega \vDash e = e'$ if $\omega\llbracket e \rrbracket = \llbracket e' \rrbracket$, and ii) $\omega \vDash \exists x P$ if there exists $r \in \mathbb{R}$ such that $\omega[x \mapsto r] \vDash P$.

HPs are defined syntactically as follows: $\alpha ::= x := e \mid x := * \mid f(\bar{x}) \mid ?P \mid x' = e \ \&\ P \mid \alpha \cup \alpha \mid \alpha; \alpha \mid \alpha^*$. The statement $x := e$ updates x to the value of e. $x := *$ updates x to a nondeterministic value. $f(\bar{x})$ refers to invoking function f with inputs $\bar{x}$. A codebase $\mathcal{C}$ maps function names to their definitions: $f(\bar{x}) = \alpha$. The statement $?P$ is a boolean test. $x' = e \ \&\ P$ is a continuous program with an ordinary differential equation $x' = e$ and an evolution domain P, where x' denotes the time derivative of x. Continuous programs are restricted to polynomial differential equations. We may specify a vector of such equations to specify a continuous evolution. We use $\alpha \cup \alpha$ for nondeterministic choice, $\alpha; \alpha$ for sequencing, and α^* for iterating α nondeterministic number of times. We show empty sequence of HPs with ϵ.

We define two HPs α_1 and α_2 structurally congruent, $\alpha_1 \equiv \alpha_2$ according to the following rules: i) Structural congruence is an equivalence relation. ii)

$\epsilon; \alpha \equiv \alpha$. iii) $\alpha^* \equiv \alpha^0 \cup \alpha^1 \cup \cdots = \cup_{i \in \mathbb{N}} \alpha^i$, where α^i is the iteration α for i times, defined as $\alpha^0 = ?\top$ and $\alpha^{i+1} = \alpha^i; \alpha$. iv) $\alpha_1 \equiv \alpha_2$ implies $\mathcal{E}[\alpha_1] \equiv \mathcal{E}[\alpha_2]$, where $\mathcal{E}$ is the evaluation context and is defined as $\mathcal{E} ::= [\,] \mid \mathcal{E}; \alpha \mid \epsilon; \mathcal{E}$.

Having defined HPs syntactically, we define a program as an HP $\alpha_{\mathfrak{p}}$ along with the codebase of all defined functions, i.e., $\mathfrak{p} = \langle \alpha_{\mathfrak{p}}, \mathcal{C} \rangle$. We assume that $\alpha_{\mathfrak{p}}$ simply invokes the specific function $main()$.

We define $\kappa ::= (t, \omega, \alpha)$ where $t \in \mathbb{R}^{\geq 0}$ is a timestamp. Accordingly, we define the timestamped operational semantics of HPs in Figure 1. According to rule O1, $x := e$ updates the state ω by mapping x to the value of e in ω. In rule O2, ω is updated with x being mapped to a nondeterministic real value. $?P$ is defined only for states that model P, without altering the state, according to rule O3. In rule O4, invoking f with inputs $\bar{e}$ runs the body of the function in a new state, where fresh parameter names are mapped to their corresponding polynomial term values. According to rule O5, $x' = e \,\&\, P$ is executable if there exists a solution for the equation $x' = e$ in the domain P. In this case, x is updated according to $x' = e$ within some nondeterministic continuous time span $[0, r]$, where every state update must satisfy the domain condition P. Rule O6 describes how $\alpha_1 \cup \alpha_2$ nondeterministically chooses between α_1 and α_2 to run. Rule O7 specifies reduction of sequences of HPs using an evaluation context, and rule O8 states that reduction follows structural congruence.

Example 1. Having introduced HPs as the source language model, let's revisit the AV example described in Section 1. We can describe the AV system as the HP $(\mathit{controller}; \mathit{machine})^*$, where *controller* is the computational component and *machine* is the physical component of the AV (the engine, braking system, etc.) defined as $\mathit{controller} = (\mathit{sen}(); (?P_1; \mathit{assess}(); (?P_2; \mathit{cmdHB}()) \cup ?\neg P_2) \cup ?\neg P_1)^*$ and $\mathit{machine} = \mathit{init}(); \mathit{evolve}()$, respectively. Function $\mathit{sen}()$ retrieves data from speed sensors and environmental sensors to detect obstacles. Predicate P_1 returns true if an obstacle is detected. Function $\mathit{assess}()$ analyzes the data to determine the threat level and appropriate response, e.g., braking level or maneuver. Predicate P_2 returns true if the threat level is high enough for hard breaking. Function $\mathit{cmdHB}()$ executes the hard braking command. In our language model, the command can be described by simply setting the acceleration parameter to a proper negative value. Function $\mathit{init}()$ assigns certain initial values for physical evolution, e.g., time and acceleration (effectively coming from $\mathit{cmdHB}()$). Finally, the function $\mathit{evolve}()$ delineates the temporal progression of the AV's position through the application of differential equations. This function is formally expressed as $k' = 1, x' = v, v' = a \,\&\, k \leq p$, where k, x, v, and a represent the time, position, velocity, and acceleration of the AV, respectively. Here, k is a continuously increasing time variable. The evolution of the AV's position, velocity, and timer is confined within its evolution domain, which imposes an upper limit on the driving duration—denoted by $k \leq p$. This constraint ensures that the AV does not operate continuously beyond a pre-set limit of p time units for safety reasons, after which control is transferred back to the *controller*.

Instantiation of $\mathit{toFOL}(\cdot)$ To logically specify a trace, we must instantiate function $\mathit{toFOL}(\cdot)$. We consider the following predicates to logically specify a trace: DAssign/3, NDAssign/2, Test/2, Call/3, Continuum/3, Choice/3, State/2, and Context/2. Note that $/n$ refers to the arity of the predicate.

We define a function to logically specify a configuration within a trace. For this purpose, we introduce the helper function $\mathit{toFOL}(\kappa)$, which returns the logical specification of κ. Let $\kappa = (t, \omega, \alpha)$. We have the following cases: i) If $\alpha \equiv \mathcal{E}[x := e]$ then $\mathit{toFOL}(\kappa) = \{\mathrm{DAssign}(t, x, e), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$. ii) If $\alpha \equiv \mathcal{E}[x := *]$ then $\mathit{toFOL}(\kappa) = \{\mathrm{NDAssign}(t, x), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$. iii) If $\alpha \equiv \mathcal{E}[?P]$ then $\mathit{toFOL}(\kappa) = \{\mathrm{Test}(t, P), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$. iv) If $\alpha \equiv \mathcal{E}[f(\bar{e})]$ then we define $\mathit{toFOL}(\kappa) = \{\mathrm{Call}(t, f, \bar{e}), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$, v) If $\alpha \equiv \mathcal{E}[x' = e \;\&\; P]$ then we define $\mathit{toFOL}(\kappa) = \{\mathrm{Continuum}(t, x' = e, P), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$. vi) Finally, if $\equiv \mathcal{E}[\alpha_1 \cup \alpha_2]$ then $\mathit{toFOL}(\kappa) = \{\mathrm{Choice}(t, \alpha_1, \alpha_2), \mathrm{Context}(t, \mathcal{E}), \mathrm{State}(t, \omega)\}$. In essence, $\mathit{toFOL}(\kappa)$ specifies the evaluation context and the redex within κ. Note that HP constructs, evaluation contexts, and states appear as predicate arguments in this presentation to enhance readability. Their syntax can be expressed as string literals to conform with the syntax of predicate logic.

We define the logical specification of traces for both finite and infinite cases based on the logical specification of configurations, using $\mathit{toFOL}(\kappa)$. Let $\tau = \kappa_0\kappa_1 \cdots \kappa_n$ for some n, then its logical specification is defined as $\mathit{toFOL}(\tau) = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i)$. Otherwise, for infinite trace $\tau = \kappa_0\kappa_1 \cdots$, we have $\mathit{toFOL}(\tau) = \bigcup_{\tau' \in \mathit{prefix}(\tau)} \mathit{toFOL}(\tau')$, where $\mathit{toFOL}(\tau') = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i)$, for $\tau' = \kappa_0\kappa_1 \cdots \kappa_n$. Showing that $\mathit{toFOL}(\tau)$ is injective and monotonically increasing can be done straightforwardly.

3.2 Instantiation of Logging Specifications

The class of logging specifications $\mathcal{LS}_{\mathit{call}}$ is defined to specify temporal relations among function invocations in HPs. Formally, $\mathcal{LS}_{\mathit{call}}$ is the set of all logging specifications LS given by $\mathit{spec}(\mathit{\Gamma}_G, \{\mathrm{LoggedCall}\})$, where $\mathit{\Gamma}_G$ is a set of Horn clauses referred to as *guidelines*, including clauses of the form

$$\begin{aligned}\forall t_0, \cdots, t_n, \bar{x}_0, \cdots, \bar{x}_n.\mathrm{Call}(t_0, f_0, \bar{x}_0) \bigwedge_{i=1}^{n} \left(\mathrm{Call}(t_i, f_i, \bar{x}_i) \wedge t_i < t_0\right) \wedge \\ \varphi(t_0, \cdots, t_n) \wedge \varphi'(\bar{x}_0, \cdots, \bar{x}_n) \implies \mathrm{LoggedCall}(f_0, \bar{x}_0),\end{aligned} \tag{1}$$

in which for all $j \in \{0, \cdots, n\}$ i) f_j is a function name with a definition in $\mathcal{C}$, ii) $\bar{x}_j$ is a placeholder for a sequence of parameters passed to f_j, and iii) $\mathrm{Call}(t_j, f_j, \bar{x}_j)$ specifies the event of invoking f_j at time t_j with parameters $\bar{x}_j$.

In (1), $\varphi(t_0, \cdots, t_n)$ represents a potentially empty conjunctive sequence of literals of the form $t_i < t_j$. Additionally, we define *triggers* and *logging events* as follows: $\mathit{Triggers}(\mathit{LS}) = \{f_1, \cdots, f_n\}$ and $\mathit{Logevent}(\mathit{LS}) = f_0$. The *logging preconditions* are predicates $\mathrm{Call}(t_i, f_i, \bar{x}_i)$ for all $i \in \{1, \cdots, n\}$.

Example 2. We can define the guideline of the logging specification for the example in Section 1 and Example 1 as follows:

$$\begin{aligned}\forall t_0, t_1, t_2, t_3, \bar{x}_0, \cdots, \bar{x}_n.\mathrm{Call}(t_0, \mathit{evolve}, \bar{x}_0) \wedge \mathrm{Call}(t_1, \mathit{cmdHB}, \bar{x}_2) \wedge t_1 < t_0 \wedge \mathrm{Call}(t_2, \mathit{assess}, \bar{x}_2) \wedge \\ t_2 < t_0 \wedge \mathrm{Call}(t_3, \mathit{sen}, \bar{x}_3) \wedge t_3 < t_0 \wedge t_3 < t_2 \wedge t_2 < t_1 \implies \mathrm{LoggedCall}(\mathit{evolve}, \bar{x}_0).\end{aligned}$$

The guidelines ensures logging the invocations of $\mathit{evolve}()$ if the functions $\mathit{sen}()$, $\mathit{assess}()$, and $\mathit{cmdHB}()$ are called in order before $\mathit{evolve}()$. In this specification, we are skipping to list a detailed collection of inputs for each function for the sake of brevity. As mentioned in Section 1, for instance, $\bar{x}_0$ could include different environmental variables, e.g., the location, speed, and acceleration.

3.3 Target HP Model

We extend $\mathcal{HP}$ to define the target program model, denoted by $\mathcal{HP}_{\log}$, with the following syntax and semantics. The task of the instrumentation algorithm is to map a program specified in $\mathcal{HP}$ to a program in $\mathcal{HP}_{\log}$.

HPs are expanded syntactically with two additional constructs: $\mathsf{callEvent}(f, \bar{e})$ and $\mathsf{emit}(f, \bar{e})$. Note that $\bar{e}$ is treated as a list of expressions rather than a sequence of them in $\mathsf{callEvent}$ and emit, ensuring they have fixed arities.

We extend the configurations in $\mathcal{HP}_{\log}$ with two additional components, as well. A configuration is defined as $\kappa ::= (t, \omega, \alpha, \Sigma, \Lambda)$, where Σ is a set of predicates of the form $\mathrm{Call}(t, f, \bar{r})$, where $f \in \mathit{Triggers}$. These preconditions are supposed to be gathered, in order to decide whether to log an event. Λ is the audit log, i.e., the set of predicates of the form $\mathrm{LoggedCall}(f, \bar{r})$. The initial configuration is $\kappa_0 = (0, \omega_0, \alpha, \emptyset, \emptyset)$.

The reduction semantics of $\mathcal{HP}_{\log}$ is given below. Note that $\mathcal{C}$ and Γ_G are part of runtime structure, but for the sake of breveity we avoid annotating each step of reduction with these static structures.

HP

$$\frac{(t, \omega, \alpha) \longrightarrow (t', \omega', \alpha')}{(t, \omega, \alpha, \Sigma, \Lambda) \longrightarrow (t', \omega', \alpha', \Sigma, \Lambda)}$$

CALL_EV

$$\frac{\Sigma' = \Sigma \cup \{\mathrm{Call}(t, f, \omega[\![\bar{e}]\!])\}}{(t, \omega, \mathsf{callEvent}(f, \bar{e}); \alpha, \Sigma, \Lambda) \longrightarrow (t, \omega, \alpha, \Sigma', \Lambda)}$$

LOG

$$\frac{\Sigma \cup \Gamma_G \vdash \mathrm{LoggedCall}(f, \omega[\![\bar{e}]\!]) \qquad \Lambda' = \Lambda \cup \{\mathrm{LoggedCall}(f, \omega[\![\bar{e}]\!])\}}{(t, \omega, \mathsf{emit}(f, \bar{e}); \alpha, \Sigma, \Lambda) \longrightarrow (t, \omega, \alpha, \Sigma, \Lambda')}$$

NO_LOG

$$\frac{\Sigma \cup \Gamma_G \nvdash \mathrm{LoggedCall}(f, \omega[\![\bar{e}]\!])}{(t, \omega, \mathsf{emit}(f, \bar{e}); \alpha, \Sigma, \Lambda) \longrightarrow (t, \omega, \alpha, \Sigma, \Lambda)}$$

CONTEXT

$$\frac{(t, \omega, \alpha, \Sigma, \Lambda) \longrightarrow (t', \omega', \alpha', \Sigma', \Lambda')}{(t, \omega, \mathcal{E}[\alpha], \Sigma, \Lambda) \longrightarrow (t', \omega', \mathcal{E}[\alpha'], \Sigma', \Lambda')}$$

$\mathcal{HP}_{\log}$ inherits the operational semantics of $\mathcal{HP}$ via rule HP. Rule CALL_EV handles reduction with the $\mathsf{callEvent}(f, \bar{e})$ statement, adding $\mathrm{Call}(t, f, \omega[\![\bar{e}]\!])$ to Σ. For the $\mathsf{emit}(f, \bar{e})$ statement, rule LOG checks if the predicate $\mathrm{LoggedCall}(f, \bar{r})$ is derivable from Σ and Γ_G. If so, it adds it to the audit log Λ; otherwise, there is no change to the log, as specified by rule NO_LOG.

The residual log of a configuration is given by $\mathit{logof}(\kappa) = \mathbb{L} = \Lambda$, where $\kappa = (\_, \_, \_, \_, \Lambda)$. This instantiation defines $\tau \rightsquigarrow \mathbb{L}$ for $\mathcal{HP}_{\log}$. Since $\mathbb{L}$ consists of logical literals, $\mathit{toFOL}(\mathbb{L}) = \mathbb{L}$ adequately specifies $\lfloor \mathbb{L} \rfloor$.

3.4 Instrumentation of HPs

In this section, we discuss the instrumentation algorithm tailored for audit logging in HPs. Next, we review how the semantic preservation can be defined for this algorithm. Lastly, we specify the main results.

Instrumentation algorithm Instrumentation algorithm $\mathcal{I}_{\mathcal{HP}}$ takes an $\mathcal{HP}$ program $\mathfrak{p} = \langle \alpha, \mathcal{C} \rangle$ and a logging specification $LS \in \mathcal{LS}_{call}$, and produces a program $\mathfrak{p}' = \langle \alpha_{\mathfrak{p}}, \mathcal{C}' \rangle$ in $\mathcal{HP}_{\mathrm{log}}$. The details of how $\mathcal{I}_{\mathcal{HP}}$ modifies the codebase $\mathcal{C}$ is given in the following. Let $\mathcal{C}(f) = [f(\bar{x}) = e]$. We have three cases: i) If $f \in \mathit{Triggers}(LS)$ then $\mathcal{C}'(f) = [f(\bar{x}) = \mathsf{callEvent}(f, \bar{x}); e]$. ii) If $f \in \mathit{Logevent}(LS)$ then $\mathcal{C}'(f) = [f(\bar{x}) = \mathsf{callEvent}(f, \bar{x}); \mathsf{emit}(f, \bar{x}); e]$. iii) If $f \notin \mathit{Triggers}(LS) \cup \mathit{Logevent}(LS)$ then $\mathcal{C}'(f) = \mathcal{C}(f)$. Intuitively, with the invocation of a function f, i) If the invocation of f serves as a trigger, its execution must be preceded by a callEvent statement. Consequently, the invocation of f is recorded in the set of logging preconditions Σ, as outlined by the rule CALL_EV. ii) If the invocation of f constitutes a logging event, its execution must also be preceded by a callEvent statement, similar to the previous scenario. Subsequently, the system evaluates whether this invocation should be logged, which is determined by the presence of an emit statement (covered by rules LOG and NO_LOG). Following this evaluation, f proceeds with its execution as usual. If the invocation of f does not serve as a trigger nor a logging event, then the function executes without any alteration in behavior.

Example 3. For the logging specification in Example 2, $\mathcal{I}_{\mathcal{HP}}$ injects statements $\mathsf{callEvent}(\mathit{sen}, \bar{x})$, $\mathsf{callEvent}(\mathit{assess}, \bar{x})$ and $\mathsf{callEvent}(\mathit{cmdHB}, \bar{x})$ to the beginning of functions $\mathit{sen}()$, $\mathit{assess}()$ and $\mathit{cmdHB}()$, respectively, as these functions are triggers to log. Since $\mathit{evolve}()$ is the logging event, $\mathcal{I}_{\mathcal{HP}}$ modifies the body of the function to be $\mathsf{callEvent}(\mathit{evolve}, \bar{x}); \mathsf{emit}(\mathit{evolve}, \bar{x}); (k' = 1, x' = v, v' = a \;\&\; k \leq p)$.

Instantiation of trace correspondence relation $:\approx$ We instantiate the abstraction of the correspondence relation $:\approx$ between source and target traces for $\mathcal{I}_{\mathcal{HP}}$. We establish the source and target trace correspondence relation as follows: $\tau_1\kappa_1 :\approx \tau_2\kappa_2$ if $\kappa_1 = (t, \omega, \alpha_1)$, $\kappa_2 = (t, \omega, \alpha_2, \Sigma, \Lambda)$, and $\mathit{trim}(\alpha_2) = \alpha_1$. The function *trim* essentially eliminates any callEvent and emit statements that $\mathcal{I}_{\mathcal{HP}}$ may introduce to an HP. *trim* can be defined as follows: i) $\mathit{trim}(\mathsf{callEvent}(f, \bar{e})) = \mathit{trim}(\mathsf{emit}(f, \bar{e})) = \epsilon$. ii) $\mathit{trim}(\alpha_1 \cup \alpha_2) = \mathit{trim}(\alpha_1) \cup \mathit{trim}(\alpha_2)$. iii) $\mathit{trim}(\alpha_1; \alpha_2) = \mathit{trim}(\alpha_1); \mathit{trim}(\alpha_2)$. iv) $\mathit{trim}(\alpha^*) = (\mathit{trim}(\alpha))^*$. v) Otherwise, $\mathit{trim}(\alpha) = \alpha$.

Main Results Main properties include two results. The instrumentation algorithm $\mathcal{I}_{\mathcal{HP}}$ is semantics preserving, and is correct. Proofs of the theorems are given in the accompanying technical report [4].

4 Related Work

Hybrid-dynamic models may use programming languages techniques to specify CPSs, particularly through HPs, as we have employed in this paper. Along with HPs, there are two other major approaches to formally model CPSs: Hybrid automata, which describe a hybrid system through a finite state transition system that captures both discrete and continuous variables in each state [3, 15], and hybrid process calculi [14, 19–21], which model CPSs in terms of agents representing physical plants and cyber components. These agents communicate through named channels, and in systems like CCPS [20], a labeled transition system demonstrates how such agents execute using shared channels.

There is a rich body of work on audit logging in CPSs as one of the necessary components of cybersecurity assurance in this domain [23]. More recent examples include developing an accountability system for autonomous robots [13], a microservices-based architecture for industrial Internet of Things and CPSs [12], and enhancing realtime CPSs with audit logging capabilities [10, 11]. These proposals have architectural approaches to audit logging, whereas we study audit logging from a programming point of view and establish formal guarantees about the quality of the generated logs in CPSs.

Information-algebraic models [17] have been used within the last decade to specify and enforce correct audit logging in different realms of computation, initially in linear functional settings [5]. Moreover, this framework has been utilized to identify and analyze direct information flows in Java-like languages [7, 27]. It has also inspired investigations into audit logging correctness in concurrent systems [6], which has facilitated the study of correct audit logging in microservices [1, 8].

5 Conclusion and Future Work

This paper introduces an algorithm designed to implement precise audit logging in CPSs by instrumenting hybrid programs according to formal audit logging requirements specified using Horn clause logic. Our approach maintains the original program's semantics, altering only audit-related operations. We prove that our algorithm consistently produces accurate audit logs, effectively avoiding unnecessary or missing logging events.

Future work will extend our audit logging research to practical environments in CPSs, adapting and testing our methodology across different programming languages and technologies to enhance its applicability and robustness for real-world use.

References

1. Ahn, N.D., Amir-Mohammadian, S.: Instrumenting microservices for concurrent audit logging: Beyond Horn clauses. In: STPSA 2022. pp. 1762–1767 (June 2022)
2. Alladi, T., Chamola, V., Zeadally, S.: Industrial control systems: Cyberattack trends and countermeasures. Computer Communications **155**, 1–8 (2020)
3. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid systems, pp. 209–229. Springer (1992)
4. Amir-Mohammadiah, S.: Technical report: Correct audit logging in hybrid-dynamic systems. Tech. rep., University of the Pacific (2024), https://sepehram.github.io/pubs/tr24.pdf

5. Amir-Mohammadian, S., Chong, S., Skalka, C.: Correct audit logging: Theory and practice. In: Principals of Security and Trust. pp. 139–162 (2016)
6. Amir-Mohammadian, S., Kari, C.: Correct audit logging in concurrent systems. Electronic Notes in Theoretical Computer Science **351**, 115–141 (September 2020)
7. Amir-Mohammadian, S., Skalka, C.: In-depth enforcement of dynamic integrity taint analysis. In: Programming Languages and Analysis for Security (2016)
8. Amir-Mohammadian, S., Zowj, A.Y.: Towards concurrent audit logging in microservices. In: STPSA 2021. pp. 1357–1362 (July 2021)
9. Ávila, R., Khoury, R., Khoury, R., Petrillo, F.: Use of security logs for data leak detection: a systematic literature review. Secur. Commun. Networks **2021**, 1–29 (2021)
10. Bansal, A., Kandikuppa, A., Chen, C.Y., Hasan, M., Bates, A., Mohan, S.: Towards efficient auditing for real-time systems. In: ESORICS. pp. 614–634. Springer (2022)
11. Bansal, A., Kandikuppa, A., Hasan, M., Chen, C.Y., Bates, A., Mohan, S.: System auditing for real-time systems. TOPS **26**(4), 1–37 (2023)
12. Dobaj, J., Iber, J., Krisper, M., Kreiner, C.: A microservice architecture for the industrial internet-of-things. In: EuroPLoP). pp. 1–15 (2018)
13. Fernández-Becerra, L., Manuel Guerrero-Higueras, Á., Rodríguez-Lera, F.J., Matellán, V.: Accountability as a service for robotics: Performance assessment of different accountability strategies for autonomous robots. Logic Journal of the IGPL **32**(2), 243–262 (2024)
14. Galpin, V., Bortolussi, L., Hillston, J.: Hype: Hybrid modelling by composition of flows. Formal Aspects of Computing **25**(4), 503–541 (2013)
15. Henzinger, T.A.: The theory of hybrid automata. In: Verification of digital and hybrid systems, pp. 265–292. Springer (2000)
16. Kim, K., Kim, J.S., Jeong, S., Park, J.H., Kim, H.K.: Cybersecurity for autonomous vehicles: Review of attacks and defense. Computers & security **103**, 102150 (2021)
17. Kohlas, J., Schmid, J.: An algebraic theory of information: An introduction and survey. Information **5**(2), 219–254 (2014)
18. Kumar, P., Lin, Y., Bai, G., Paverd, A., Dong, J.S., Martin, A.: Smart grid metering networks: A survey on security, privacy and open research issues. IEEE Communications Surveys & Tutorials **21**(3), 2886–2927 (2019)
19. Lanese, I., Bedogni, L., Di Felice, M.: Internet of things: a process calculus approach. In: SAC. pp. 1339–1346 (2013)
20. Lanotte, R., Merro, M.: A calculus of cyber-physical systems. In: ICALP. pp. 115–127. Springer (2017)
21. Lanotte, R., Merro, M.: A semantic theory of the internet of things. Information and Computation **259**, 72–101 (2018)
22. Lehto, M.: Cyber-attacks against critical infrastructure. In: Cyber security: Critical infrastructure protection, pp. 3–42. Springer (2022)
23. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Computing Surveys (CSUR) **46**(4), 1–29 (2014)
24. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41**(2), 143–189 (2008)
25. Platzer, A.: The complete proof theory of hybrid systems. In: 2012 27th Annual IEEE Symposium on Logic in Computer Science. pp. 541–550. IEEE (2012)
26. Platzer, A.: Logical foundations of cyber-physical systems, vol. 662. Springer (2018)
27. Skalka, C., Amir-Mohammadian, S., Clark, S.: Maybe tainted data: Theory and a case study. Journal of Computer Security **28**(3), 295–335 (April 2020)