Instrumenting Microservices for Concurrent Audit Logging: Beyond Horn Clauses

Nicolas Duri Ahn
Dept. of Computer Science
University of the Pacific
Stockton, CA, USA
n_ahn2@u.pacific.edu

Sepehr Amir-Mohammadian
Dept. of Computer Science
University of the Pacific
Stockton, CA, USA
samirmohammadian@pacific.edu

***Abstract*—Instrumenting legacy code is an effective approach to enforce security policies. Formal correctness of this approach in the realm of audit logging relies on semantic frameworks that leverage information algebra to model and compare the information content of the generated audits logs and the program at runtime. Previous work has demonstrated the applicability of instrumentation techniques in the enforcement of audit logging policies for systems with microservices architecture. However, the specified policies suffer from the limited expressivity power as they are confined to Horn clauses being directly used in logic programming engines. In this paper, we explore audit logging specifications that go beyond Horn clauses in certain aspects, and the ways in which these specifications are automatically enforced in microservices. In particular, we explore an instrumentation tool that rewrites Java-based microservices according to a JSON specification of audit logging requirements, where these logging requirements are not limited to Horn clauses. The rewritten set of microservices are then automatically enabled to generate audit logs that are shown to be formally correct.**

***Index Terms*—Audit logs, concurrent systems, microservices, programming languages, security**

I. Introduction

Audit logging is a prevalent mechanism used to capture the runtime events for the purpose of post-facto analysis, user accountability, diagnostics, security in-depth, etc. In this regard, enabling systems to generate necessary and sufficient logs plays a crucial role to meet the goals of audit logging. However, inadequate audit logging has been recognized as common problem in software development [**?**], [1]. While the audit log inadequacy problem can be solved by naively recording massive volume of information at runtime, this approach incurs inefficiency in performance and response to different security incidents [3]. For example, industrial control systems may suffer from the lack of monitoring audit logs in realtime to identify the security breaches [**?**]. Correctness of audit logging ensures to only record the necessary information.

Information-algebraic [4] models have been used within the last decade to provide semantic frameworks for audit logging. This has been accomplished by interpreting audit logs as well as the runtime structure of processes as information-algebraic elements, intuitively reflecting on their information content [**?**]. Correct audit logging relies on this algebraic interpretation by comparing the information content of audit logs vs. the programs at runtime. Using this semantic framework, an implementation model has been proposed for linear processes that ensures correct audit logging, leveraging program instrumentation techniques [5]. The same framework has also been employed to identify and analyze direct information flows in Java-like settings [**?**], [**?**]. This semantic framework has also inspired the study of audit logging correctness in concurrent systems [6], which facilitates to log an event if one or more trigger events have occurred on the same or other concurrent components. The audit logging requirements can be specified with Horn clauses, where each event is a call to a function in a concurrent component, associated with its contextual information, e.g., the time of occurrence.

Using microservices has become a popular approach in application development in recent years, and many organizations report success in its adoption [**?**]. Some surveys show more than 70% of partial or full adoption worldwide in recent years [**?**]. In this approach, a system is decomposed into multiple standalone loosely-coupled components, called microservices. Each microservice has its own database, and can run on a separate machine, VM, or container. Microservices commonly communicate with each other through RESTful APIs. The accommodated modularity by a microservices-based system provides better maintainability which results in improved security, feature updates, and the ability to continue operating (at least partially) despite the failure of one or more components. Microservices-based applications are architecturally concurrent and thus are good candidates to study the effectiveness of the aforementioned framework for audit logging in concurrent environments.

Using the implementation model for audit logging in concurrent systems [6], in a previous work [**?**] we have described an instrumentation tool for Java microservices that are built on Spring framework. The tool supports audit logging requirements that can be specified by Horn clauses, according to which the instrumentation tool modifies different microservices so that concurrent audit logging is supported by the whole system. To accomplish this, the instrumented services may contact a Prolog engine to communicate the Horn clause specification of the logging requirements as well as the trigger and logging events. The Prolog engine is used to deduce whether logging in a certain microservices must take place.

In this paper, we go beyond Horn clauses to specify audit

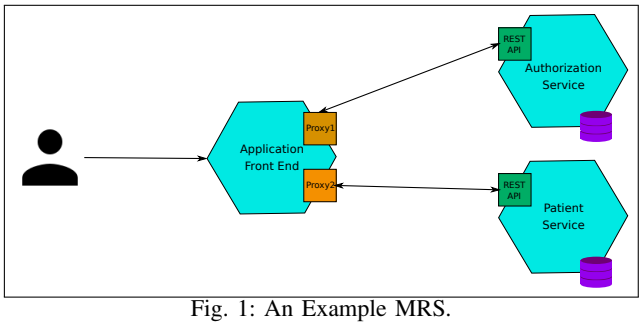

Fig. 1: An Example MRS.

logging requirement and study a sequel to the instrumentation tool, mentioned above, where we can use a more expressive class of audit logging requirements to specify the conditions upon which logging must happen. The formal implementation model of correct audit logging in concurrent systems with the extended audit logging requirements is described in [**?**]. Our proposed tool for Java microservices relies on this implementation model. Since audit logging requirements are extended beyond Horn clauses our tool cannot simply communicate the logging requirements directly to a Prolog engine. We propose an algorithm through which Prolog is used to deduce intermediary candidate events to be logged and filter out the ones that do not satisfy logging preconditions.

In the following, we describe an illustrative example that demonstrates the need to go beyond Horn clauses to specify audit logging requirements.

Example: Microservices-based medical records system. Let's consider a medical records system (MRS) with microservices architecture. Among numerous services that such system includes, this example assumes the existence of at least the following: 1) A front-end service that authenticates the users and multiplexes their queries to the back-end services, 2) an authorization system that controls access to system resources, and 3) a patient services that manages patient information. Figure 1 demonstrates these microservices in a high-level manner. One common authorization-related operation in healthcare systems is to "break-the-glass" [8] in critical scenarios, which refers to accessing certain information (e.g., patient medical history) without any mediating authorization checks. After a user breaks the glass, actions of that user is recorded in the log for a posteriori analysis and establish user accountability. A first attempt to specify the policy could be *"Store in the log all accesses to patient medical history by a user who has already broken the glass"*. This policy can be specified in Horn clause logic, and for which previous work proposes an implementation model [6] and a tool for Java-based microservices [**?**].

However, the aforementioned policy entails to log all such accesses indefinitely. In reality, breaking the glass is deactivated once the critical situation is resolved. Therefore, we may consider some dual capability "mend-the-glass" that restores access control and enforces it fully, if the glass is already broken. Having this capability, the example break-the-glass policy can be restated as follows: *"Store in the log all accesses to patient medical history by a user who has already broken the glass and has not mended it afterwards"*. This updated more realistic specification goes beyond Horn clauses in expressivity.

Therefore, a compelling next step is to consider a more powerful instrumentation tool to implements logging specifications that support extended preconditions like the one discussed above. Our instrumentation tool relies on implementation models of correct audit logging which support more expressive classes of logging specifications.

Paper outline. The rest of the paper is organized as follows. In Section II, we review the formal implementation model, discussing an instrumentation algorithm for systems in π-calculus. In Section III, we discuss our tool to instrument microservices in Java Spring. In addition, we present a demo of a microservices-based MRS, and its instrumentation by our tool. Related work is discussed in Section IV. Finally, Section V concludes the paper.

II. A Brief Review of the Instrumentation Model

In [**?**], we have explored the full formalization of the implementation model for instrumenting concurrent systems that guarantee the correctness of audit logs, according to logging specifications that go beyond Horn clauses. In this section, we review this model briefly.

A. Source System Model

The source system Π is concurrent program, which is modeled in π-calculus [9]. Top-level components of Π, denoted by A, are called top-level agents. Top-level agents execute in parallel, and communicate among themselves as their functionality dictates. A consists of internal modules and/or functions, modeled as subagents B^A. As part of the definition of the source system, we assume the existence ofß codebases $\mathcal{C}_{\mathcal{U}}$ and $\mathcal{C}_{\mathcal{L}}$ that include definitions of the form $A(x_1, \cdots, x_n) \triangleq \cdots$ and $B^A(x_1, \cdots, x_n) \triangleq \cdots$ resp.

B. A Class of Logging Specifications

Horn clause logic has been used to specify audit logging requirements in previous work [**?**], where certain events must take place so that audit logs are generated. In this paper, we go beyond Horn clauses in our implementation model to help specify not only the necessity of having certain events to occur, but also to ensure that another group of events do not take place. In this respect, we define a class of specifications that assert temporal relations among the events that must or must not transpire in different components (top-level agents) of the system. Using this extended class of specifications, a particular event must be logged, provided that a certain set of events have taken place and another set of events have not. We would call these different sets of events positive and negative triggers, resp. We use $Spec_{log}$ to denote this class of specifications. In $Spec_{log}$, each event is modeled as a subagent invocation, i.e., a module within one of the agents of the system. Figure 2 depicts the structure of specifications in $Spec_{log}$. $\mathrm{Call}(t, A, B, xs)$ asserts the event of invoking subagent B^A at time t with list of parameters xs. φ and ψ'_j are

$$\forall t_0, \cdots, t_n, xs_0, \cdots, xs_n \,.\, \mathrm{Call}(t_0, A_0, B_0, xs_0) \bigwedge_{i=1}^{n} (\mathrm{Call}(t_i, A_i, B_i, xs_i) \wedge t_i < t_0) \wedge \varphi(t_0, \cdots, t_n) \wedge \varphi'(xs_0, \cdots, xs_n) \bigwedge_{j=1}^{m} (\forall t'_j, ys_j \,.\, \psi_j(xs_0, \cdots, xs_n, ys_j) \wedge \psi'_j(t_0, \cdots, t_n, t'_j) \implies \neg\mathrm{Call}(t'_j, A'_j, B'_j, ys_j)) \implies \mathrm{LoggedCall}(A_0, B_0, xs_0)$$

Fig. 2: $Spec_{log}$ clause structure.

$$\mathcal{C}_{\mathcal{L}}(\texttt{Authorization})(\texttt{breakTheGlass}) = [\texttt{breakTheGlass}^{\texttt{Authorization}} \triangleq P] \text{ for some } P$$

$$\mathcal{C}_{\mathcal{L}}(\texttt{Authorization})(\texttt{mendTheGlass}) = [\texttt{mendTheGlass}^{\texttt{Authorization}} \triangleq P'] \text{ for some } P'$$

$$\mathcal{C}_{\mathcal{L}}(\texttt{Patient})(\texttt{getMedicalHistory}) = [\texttt{getMedicalHistory}^{\texttt{Patient}} \triangleq Q] \text{ for some } Q$$

$$\forall t_0, t_1, p, u \,.\, \mathrm{Call}(t_0, \texttt{Patient}, \texttt{getMedicalHistory}, [p, u]) \wedge \mathrm{Call}(t_1, \texttt{Authorization}, \texttt{breakTheGlass}, [u]) \wedge t_1 < t_0 \wedge (\forall t_2 \,.\, t_2 < t_0 \wedge t_1 < t_2 \implies \neg\mathrm{Call}(t_2, \texttt{Authorization}, \texttt{mendTheGlass}, [u])) \implies \mathrm{LoggedCall}(\texttt{Patient}, \texttt{getMedicalHistory}, [p, u])$$

Fig. 3: Example logging specification for an MRS.

possibly empty conjunctive sequence of literals of the form $t_i < t_j$. A_0 is called a *logging event agent*, whereas other A_is are called *positive trigger agents*. A'_js are called *negative trigger agents*. Similarly, *logging event sub-agent* refers to B_0, and other B_is are called *positive trigger sub-agents*. B'_js are called *negative trigger sub-agents*. *Logging preconditions* are predicates $\mathrm{Call}(t_i, A_i, B_i, \tilde{x})$ for all $i \in \{1, \cdots, n\}$ and $\mathrm{Call}(t_j, A'_j, B'_j, ys_j)$ for all $j \in \{1, \cdots, m\}$.

For example, in the microservices-based MRS, described in Figure 1, each microservice, including Authorization and Patient, is a top-level agent in Π. Authorization may include modules to break and mend the glass, and Patient may have a functionality to read patient medical history. These functionalities are defined as part of $\mathcal{C}_{\mathcal{L}}$ (Figure 3). Moreover, Figure 3 describes the logging specification in $Spec_{log}$ that is associated with the break-the-glass policy. In this logical specification, t_0, t_1 and t_2 are timestamps, and t_1 precedes t_0. p refers to the patient identifier, and u is the user identifier who breaks the glass and attempts to read the medical history of p later on. Additionally, logging is preconditioned on the fact that the glass is not mended in between the events of breaking the glass and gaining access to the patient medical data. t_2 is the timestamp of the negative trigger, i.e., mend-the-glass operation.

C. Target System Model

The instrumentation algorithm maps a Π system to a target system, denoted by $\Pi_{\log}$ system. Runtime environment of $\Pi_{\log}$ includes a timing counter t, Δ that returns the set of logging preconditions transpired in a given agent, Σ that returns the set of logging preconditions that have transpired in the triggers of a given agent, and Λ that stores the audit logs for a given agent. The preconditions in Σ must be communicated between a given agent and all triggers agents. Certain prefixes are added to $\Pi_{\log}$ to facilitate storage and retrieval of information to these runtime components: 1) $\mathsf{callEvent}(A, B, \tilde{x})$ updates $\Delta(A)$ with predicate $\mathrm{Call}(t, A, B, \tilde{x})$, 2) $\mathsf{addPrecond}(x, A)$ updates$\Sigma(A)$ with precondition x, 3) $\mathsf{sendPrecond}(x, A)$ converts $\Delta(A)$ to a transferable object and sends it though link x, and 4) $\mathsf{emit}(A, B, \tilde{x})$ studies the derivability of $\mathrm{LoggedCall}(A, B, \tilde{x})$ and accordingly $\Lambda(A)$ is updated with this predicate.

$$\mathcal{C}'_{\mathcal{L}}(\texttt{Authorization})(\texttt{breakTheGlass}) = [\texttt{breakTheGlass}^{\texttt{Authorization}}(u) \triangleq \mathsf{callEvent}(\texttt{Authorization}, \texttt{breakTheGlass}, [u]).P] \qquad \mathcal{C}'_{\mathcal{L}}(\texttt{Authorization})(\texttt{mendTheGlass}) = [\texttt{mendTheGlass}^{\texttt{Authorization}}(u) \triangleq \mathsf{callEvent}(\texttt{Authorization}, \texttt{mendTheGlass}, [u]).P']$$

$$\mathcal{C}'_{\mathcal{L}}(\texttt{Patient})(\texttt{getMedicalHistory}) = [\texttt{getMedicalHistory}^{\texttt{Patient}}(p, u) \triangleq \mathsf{callEvent}(\texttt{Patient}, \texttt{getMedicalHistory}, [p, u]).c_{\overline{\texttt{PA}}}.c_{\texttt{PA}}(f).\mathsf{addPrecond}(f, \texttt{Patient}).\mathsf{emit}(\texttt{Patient}, \texttt{getMedicalHistory}, [p, u]).Q] \qquad \mathcal{C}'_{\mathcal{L}}(\texttt{Authorization})(D_{\texttt{PA}}) = [D_{\texttt{PA}}^{\texttt{Authorization}}(c_{\texttt{PA}}) \triangleq c_{\texttt{PA}}.\mathsf{sendPrecond}(c_{\texttt{PA}}, \texttt{Authorization}).D_{\texttt{PA}}^{\texttt{Authorization}}(c_{\texttt{PA}})]$$

Fig. 4: Example Instrumentation of the MRS.

D. Instrumentation Algorithm

Given a logging specification in $LS \in Spec_{log}$, algorithm $\mathcal{I}$ translates a Π-system into a $\Pi_{\log}$-system with concurrent audit logging capabilities to support LS. In the following we briefly point out main aspects of how $\mathcal{I}$ works.

If A_i is a logging event agent and A_j is a positive or negative trigger agent, a fresh link c_{ij} is added between them. This link is used to communicate locally transpired trigger events that are stored in trigger agents' Δ component. sendPrecond and addPrecond prefixes are used for this purpose.

$\mathcal{I}$ inserts prefix callEvent before at the starting point of each negative/positive trigger sub-agent. This ensures storing trigger events in Δ.

callEvent is also inserted at the starting point of logging event sub-agents to capture logging events in Δ component of the corresponding agents. In addition, addPrecond is inserted to notify all positive/negative trigger agents to send their trigger events through the established c_{ij} link to the logging event tigger. The received trigger events are then stored in Σ component of the logging event agent. Finally, the logging event must check if the invocation of logging event sub-agent will be logged in Λ. For this purpose, emit is inserted.

Each positive/negative trigger A_j must be able to respond to the requests by some logging event agent A_i on the c_{ij} link. To this end, $\mathcal{I}$ introduces new sub-agent D_{ij} in code base of A_j that indefinitely listens on that link and returns the content of Δ on the same link. This is accomplished by sendPrecond prefix.

As an example, having the definitions in Figure 3 for an MRS, $\mathcal{I}$ modifies the definition of sub-agents $\texttt{breakTheGlass}^{\texttt{Authorization}}$, $\texttt{mendTheGlass}^{\texttt{Authorization}}$ and $\texttt{getMedicalHistory}^{\texttt{Patient}}$ by injecting proper prefixes at their starting points. In addition, a new link $c_{\texttt{PA}}$ is introduced between `Authorization` and `Patient` agents, and a new sub-agent $D_{\texttt{PA}}^{\texttt{Authorization}}$ is added that listens on $c_{\texttt{PA}}$ indefinitely for requests from `Patient` and responds with locally transpired trigger events, i.e., invocations to `breakTheGlass` (positive) and `mendTheGlass` (negative) in this example.

III. Instrumenting Microservices for Audit Logging

In this section, we discuss the implementation details of the instrumentation algorithm described in Section II, customized for microservices-based applications. This tool, `LogInst.v2`, is an extension of `LogInst` described in []. Our review of `LogInst.v2` is followed by a demo on our medical records example. `LogInst.v2` (similar to

LogInst) treats each microservice as an agent of the concurrent application. Moreover, each method of a microservice plays the role of a sub-agent of the agent represented by that microservice.

A. LogInst

In what follows we briefly describe how LogInst works. The reader is referred to [] for more details. LogInst receives logging specification in JSON format along with the source code of microservices written in Java Spring framework, and applies certain modifications to those microservices such that the audit logging is supported correctly according to the formal specification. LogInst parses JSON specifications that lack negative triggers and are translatable to Horn clauses. In this regard, the logging event microservice is modified to launch SWI Prolog [] engine in parallel to its main process. The logging event microservice communicates the logging specifications with the engine, and queries the engine to infer whether certain events must be logged.

LogInst may add one or more repositories to a microservice. There are three different types of repositories: 1) local-db stores local trigger or logging events in a microservice, 2) remote-db stores all trigger events associated with a logging specification, and 3) log-db stores the log. A trigger microservice may have access to local-db only, whereas a logging event microservice acquires all three types of repositories. local-db , remote-db, log-db implement Δ, Σ, and Λ, resp.

LogInst relies on AspectJ, in particular *before* aspects, to extend the functionalities of trigger and logging event microservices for logging purposes. Both trigger and logging event methods are preceded by storing the event of invoking that method in local-db. In addition, each logging event method queries all trigger microservices to send the content of their local-db and aggregates them in its remote-db. Finally, the logging event microservice queries the Prolog engine for all inferred logs and adds them in its log-db In order to query the logic engine, the logging event microservice communicates the contents of local-db and remote-db with the engine first. These interactive steps implement callEvent, addPrecond, and emit prefixes in $\mathcal{I}$[1].

As mentioned above, a logging event microservice needs to query all trigger microservices for the content of their local-db repository. This is facilitated by LogInst through the addition of a REST controller in each trigger microservices that listens indefinitely on a specific URL for incoming queries from the logging event microservice. In addition, LogInst extends the logging event microservice with a web client to send such queries to the trigger microservices asynchronously.

[1]Note that emit in LogInst is limited to querying Prolog based on Horn clause specifications of audit logging only, where as emit described in Section **??** goes beyond Horn clauses.

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

Fig. 5: Intermediary clause structure.

$$\forall t_0, \cdots, t_n, xs_0, \cdots, xs_n, t'_j, ys_j \,.\, \psi_j(xs_0, \cdots, xs_n, ys_j) \wedge \psi'_j(t_0, \cdots, t_n, t'_j) \wedge \mathrm{Call}(t'_j, A'_j, B'_j, ys_j) \implies \mathrm{NegativeTrigger}(j, t_0, \cdots, t_n, xs_0, \cdots, xs_n)$$

Fig. 6: Negative trigger clause.

B. LogInst.v2

LogInst.v2 instruments microservices similar to LogInst to a great extent. In what follows, we describe the major ways in which LogInst.v2 differs from LogInst to support more expressive logging specifications.

LogInst.v2 treats negative trigger microservices similar to positive ones, i.e., each negative trigger is extended with local-db repository, and upon transpiring a negative trigger event, that event is stored in that repository. Moreover, each negative trigger microservice is extended with a REST controller, similar to positive trigger microservices, that respond with the content of the local-db repository.

The main difference of LogInst.v2 with the previous version is that in contrast to LogInst, LogInst.v2 cannot translate logging specifications to Horn clauses and communicate them with the Prolog engine, due to higher expressivity of the specifications. These more expressive specifications are still passed to the instrumenting tool in JSON. LogInst.v2 parses the JSON specification which specifies logical rules of the form given in Figure 2. LogInst.v2 instruments the logging event microservices to initially redacts the negative triggers and builds intermediary Horn clauses of the form depicted in Figure 5. These clauses are communicated with the Prolog engine.

Logging event microservice is instrumented in the same style as LogInst to add local event to local-db, and reach out to (negative and positive) trigger microservices to store trigger events in its remote-db. However, the next few steps differ from the one supported by LogInst.

In order to decide whether an event must be logged by a logging event microservice, the service sends query of the form $\mathrm{LG}(A_0, B_0, t_0, \cdots, t_n, xs_0, \cdots, xs_n)$ to the Prolog engine. The result is temporarily stored in a container. Let's call this data strucuture lg-list. Next, the microservice checks whether $\mathrm{LoggedCall}(\mathrm{A}_0, \mathrm{B}_0, xs_0)$ is derivable by studying the preconditions of every negative trigger. In this regard, for each negative trigger microservice and method pair (A'_j, B'_j) (of Figure 2) it adds the clause given in Figure 6 to the Prolog engine. The logging event microservice goes through each candidate in lg-list and checks whether $\mathrm{NegativeTriggers}$ are derivable. The algorithm in Figure 7 describes logging event's behavior to infer whether $\mathrm{LoggedCall}(\mathrm{A}_0, \mathrm{B}_0, xs_0)$ is derivable. Note that LogInst.v2 does not hard code any of the negative trigger preconditions in the instrumented logging event microservice. These preconditions are read from JSON specification, translated to Horn clauses of the form given in Figure 6, and communicated with the Prolog engine.

```
for each candidate LG(A0, B0, t0, ..., tn, xs0, ..., xsn) in lg-list:
  if NegativeTrigger(1, t0, ..., tn, xs0, ..., xsn) is derivable:
    LoggedCall is not derivable. Continue with the next candidate.
  if NegativeTrigger(2, t0, ..., tn, xs0, ..., xsn) is derivable:
    LoggedCall is not derivable. Continue with the next candidate.
  ...
  if NegativeTrigger(m, t0, ..., tn, xs0, ..., xsn) is derivable:
    LoggedCall is not derivable. Continue with the next candidate.
  LoggedCall is derivable. Add LoggedCall(A0,B0,xs0) to log-db.
```

Fig. 7: Pseudocode of LoggedCall inference.

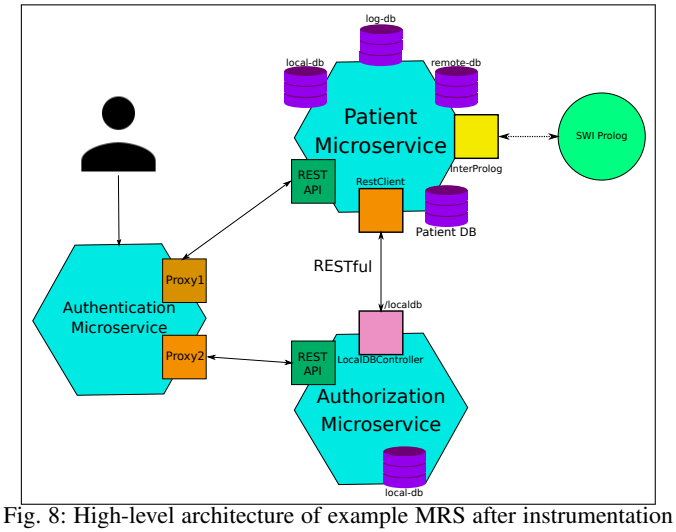

Fig. 8: High-level architecture of example MRS after instrumentation by `LogInst` (and `LogInst.v2`) using the specification of Figure **??**.

C. Instrumenting example MRS with `LogInst.v2`

Section I discusses an oversimplified MRS consisting of several microservices. In this section, we describe how `LogInst.v2` instruments this system according to the logging specification given in Figure **??**. Structurally `LogInst.v2` modifies the microservices by adding `local-db` repository to the Authorization microservice (trigger), and all three types of repositories to the Patient microservice (logging event). In addition, Patient service runs Prolog engine and communicates with it different logical rules and facts. Authorization microservice is extended with a REST controller that responds to the Patient-side web client HTTP Get requests about its `local-db` content. These architectural changes are depicted in Figure 8. The reader is referred to [] for details on how these changes are applied. We mainly focus on the aspects that differentiates the rewritten application from `LogInst`'s output in this section.

`LogInst.v2` modifies Patient service to create intermediary clause given in Figure 9 and communicates it with Prolog engine. Note that in this rule, methods' full package/class path is redacted for brevity. Moreover, since there is a single negative trigger in this example, a single negative trigger clause is created by the Patient service that can be used to ensure whether the preconditions are met for that negative trigger. This rule is also communicated with the Prolog engine.

Upon deciding to log an event, Patient service queries Prolog for all instances of `lg(patient-service, getPatientMedHistory,T0,T1,[U,P])` and stores the result in `lg-list`. Then, Patient service follows the pseudocode in Figure 10 to infer whether logging events must be logged.

```
/* Intermediary Clause */
lg(patient-service, getPatientMedHistory, T0,T1,[U,P]) :-
  funccall(T0, patient-service, getPatientMedHistory, [U, P]),
  funccall(T1, authorization-service, breakTheGlass, [U]),
  <(T1, T0),
  ==(U, user).

/* Negative Trigger Clause */
negative_trigger(T0,T1,[U,P]) :-
  funccall(T2, authorization-service, mendTheGlass, [U]),
  <(T2, T0),
  <(T1, T2).
```

Fig. 9: Intermediary clause for the example specification in Figure **??**.

```
for each candidate lg(patient-service, getPatientMedHistory,T0,T1,[U,P])} in lg-list:
  if negative_trigger(T0,T1,[U,P]) is derivable:
    LoggedCall is not derivable. Continue with the next candidate.
  LoggedCall is derivable. Add LoggedCall(patient-service, getPatientMedHistory,[U,P]) to log-db.
```

Fig. 10: Example pseudocode of LoggedCall inference.

IV. Related Work

Audit logging in microservices. In recent years, constructing software in terms of decoupled microservices [13]–[16] has been a trending approach in web application design and deployment, and thus different studies have been conducted on microservices security [17]–[19]. In practice, enforcing in-depth security has pushed platform-specific monitoring and logging techniques for microservices, e.g., in Azure Kubernetes Service [20] and Spring Security Framework [21], [22]. One common approach has been to establish a central logging service with data visualization capabilities [23]. Examples include a provenance logger for microservices-based applications [24], and an architecture for IoT services that includes logger microservices in Web of Objects platform [25]. Our approach in audit logging is concurrent rather than central, i.e., any microservice is able to log events based on preconditions that may occur in other microservices as well as that microservice. This boosts the expressivity of the enforcible logging policies. There have been other approaches to define semantics of microservices, including Petri nets [26].

Formal study of audit logging. One line of work wrt formal study of audit logging focuses on the security of logs, in particular through cryptographic techniques, e.g., to establish forward secrecy [27], to ensure trustworthiness of logs [28], [29], and to preserve privacy in auditing [30]. These techniques assume that logs are given in the first place to be secured. However, in this paper we aim at developing a tool to generate audit logs according to a provably correct model, and thus security of the logged data is orthogonal to it.

Another line of work uses logical frameworks to establish accountability in access to system resources. Examples include a framework to enforce accountability goals in discretionary access control [31], accountability wrt access to personal information based on owner-defined usage policies [32], distributed accountability based on turn-based games [33], and logging the proof of having access to system resources [34], [35]. Another related area of work is the language-level analysis of generated audit logs [36], [37].

Correct audit logging. Information algebra [4] has been used to describe the semantics of audit logging [5] for linear process execution that defines notion of correctness for audit

logs, along with an instrumentation model that guarantees to generate correct audit logs. Lately, an instrumentation model has been proposed for concurrent systems based on the information-algebraic semantic framework [6]. This model enjoys correct audit logging, which has been the basis for our proposed instrumentation tool.

Provenance. Audit logging is closely associated with the notion of provenance tracking [38]–[40]. Recent works in this area include ClearScope [41] a provenance tracker for Android devices, CamFlow [42] an auditing and provenance capture utility in Linux, and AccessProv [43] an instrumentation tool to discover vulnerabilities in Java applications.

V. Conclusion

In this paper, we have proposed a tool, `LogInst`, to instrument microservices-based applications that are deployed in Java Spring Framework for audit logging purposes. Our tool is based on an implementation model for concurrent systems that guarantees correctness of audit logging, using an information-algebraic semantic framework. `LogInst` receives the application source code, consisting of two or more microservices, along with a specification of audit logging requirements in JSON format. `LogInst` parses the JSON specification and extracts Horn clauses that are fed to a logic programming engine. `LogInst` instruments the microservices according to this specification. The instrumentation includes adding new repositories to the corresponding microservices, extending RESTful APIs on those microservices for logging-related communications, and weaving audit logging into the control flow of microservices using AspectJ. Our case study is a medical records system in which certain actions in authorization microservice may trigger logging events in access to patient medical data.

References

[1] "Top 10-2017 A10-Insufficient Logging & Monitoring," https://rb.gy/mj1xpf, 2017, accessed: 2021-03-05.

[2] "CWE-778: Insufficient Logging," https://rb.gy/2hhb5o, 2021, accessed: 2021-04-07.

[3] "CWE-779: Logging of Excessive Data," https://rb.gy/myvjgc, 2021, accessed: 2021-04-07.

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

[5] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Correct audit logging: Theory and practice," in *POST*, 2016, pp. 139–162.

[6] S. Amir-Mohammadian and C. Kari, "Correct audit logging in concurrent systems," *To appear in ENTCS*, 2020, as part of the Proceedings of LSFA.

[7] "Global Microservices In Healthcare Market Will Reach USD 519 Million By 2025," https://rb.gy/pi8y7l, 2019, accessed: 2021-04-07.

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

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

[10] "SWI Prolog," https://www.swi-prolog.org/, accessed: 2021-04-15.

[11] M. Calejo, "Interprolog: Towards a declarative embedding of logic programming in java," in *JELIA*. Springer, 2004, pp. 714–717.

[12] 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.

[13] 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.

[14] 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.

[15] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Software*, vol. 146, pp. 215–232, 2018.

[16] J. Salibindla, "Microservices api security," *Int. J. Eng. Res.*, vol. 7, no. 1, pp. 277–281, 2018.

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

[18] P. Nkomo and M. Coetzee, "Software development activities for secure microservices," in *ICCSA*. Springer, 2019, pp. 573–585.

[19] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, "Securing microservices," *IT Professional*, vol. 21, no. 1, pp. 42–49, 2019.

[20] M. Wasson, "Monitoring a microservices architecture in Azure Kubernetes Service (AKS)," https://rb.gy/xzlm95, 2020, accessed: 2021-04-15.

[21] Q. Nguyen and O. Baker, "Applying spring security framework and oauth2 to protect microservice architecture api," *Journal of Software*, pp. 257–264, 2019.

[22] O. Baker and Q. Nguyen, "A novel approach to secure microservice architecture from owasp vulnerabilities," in *CITRENZ (2019)*.

[23] J. Kazanavičius and D. Mažeika, "Migrating legacy software to microservices architecture," in *eStream*. IEEE, 2019, pp. 1–5.

[24] W. Smith, T. Moyer, and C. Munson, "Curator: provenance management for modern distributed systems," in *TaPP*, 2018.

[25] M. A. Jarwar, S. Ali, M. G. Kibria, S. Kumar, and I. Chong, "Exploiting interoperable microservices in web objects enabled internet of things," in *ICUFN*. IEEE, 2017, pp. 49–54.

[26] M. Camilli, C. Bellettini, L. Capra, and M. Monga, "A formal framework for specifying and verifying microservices based process flows," in *SEFM*. Springer, 2017, pp. 187–202.

[27] A. A. Yavuz and P. Ning, "BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems," in *ACSAC*, 2009, pp. 219–228.

[28] 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.

[29] R. Accorsi, "Bbox: A distributed secure log architecture," in *EuroPKI*, 2010, pp. 109–124.

[30] A. J. Lee, P. Tabriz, and N. Borisov, "A privacy-preserving interdomain audit framework," in *WPES*, 2006, pp. 99–108.

[31] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini, "Audit-based compliance control," *Int. J. Inf. Secur.*, vol. 6, no. 2-3, pp. 133–151, 2007.

[32] 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.

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

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

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

[36] F. Bavera and E. Bonelli, "Justification logic and audited computation," *J. Log. Comput*, vol. 28, no. 5, pp. 909–934, 2015.

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

[38] W. Ricciotti, "A core calculus for provenance inspection," in *PPDP*. ACM, 2017, pp. 187–198.

[39] M. Herschel, R. Diestelkämper, and H. B. Lahmar, "A survey on provenance: What for? what form? what from?" *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.

[40] P. Buneman and W.-C. Tan, "Data provenance: What next?" *ACM SIGMOD Record*, vol. 47, no. 3, pp. 5–16, 2019.

[41] M. Gordon, J. Eikenberry, A. Eden, J. Perkins, and M. Rinard, "Precise and comprehensive provenance tracking for android devices," Tech. Rep., 2019.

[42] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *SoCC*. ACM, 2017, pp. 405–418.

[43] F. Capobianco, C. Skalka, and T. Jaeger, "ACCESSPROV: Tracking the provenance of access control decisions," in *TaPP*, 2017.