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], [2]. 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 [4]. Correctness of audit logging ensures to only record the necessary information.

Information-algebraic [5] 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 [6]. The same framework has also been employed to identify and analyze direct information flows in Java-like settings [7], [8]. This semantic framework has also inspired the study of audit logging correctness in concurrent systems [9], 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 [10]. Some surveys show more than 70% of partial or full adoption worldwide in recent years [11]. 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.

Based on the implementation model for audit logging in concurrent systems [9], in a previous work [12] 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 microservice must take place.

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

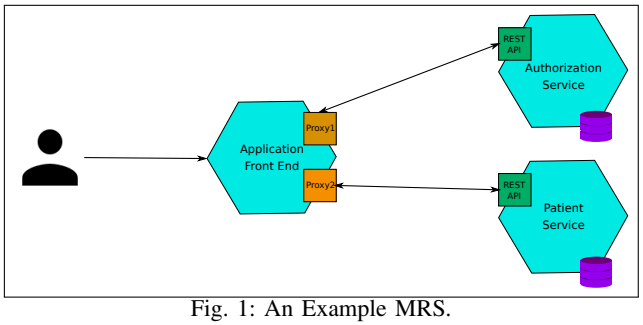

Fig. 1: An Example MRS.

logging requirements and study a sequel to the aforementioned instrumentation tool, 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 elsewhere [13]. 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 of such system, 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 service 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" [14] 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 [9] and a tool for Java-based microservices [12].

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 implement logging specifications that support extended preconditions like the one discussed above. Our instrumentation tool relies on 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. Section IV disucsses the related work and concludes the paper.

II. A Brief Review of the Instrumentation Model

In [13], we have explored the full formalization of the implementation model for instrumenting concurrent systems that guarantees 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 a concurrent program, modeled in π-calculus [15]. 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 of 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 [12], 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 possibly empty conjunctive sequence of literals of the form

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

Fig. 2: Structure of a $Spec_{log}$ logging specification.

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

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

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

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

Fig. 3: Example logging specification for an MRS.

$t_i < t_j$. φ' and ψ_j are possibly empty conjunctive sequence of literals as well. 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 at a later time. 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 $\Pi_{\log}$. 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 $LS \in Spec_{log}$, 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 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.

Having the definitions of Figure 3 for an example MRS, Figure 4 demonstrates the changes applied by the instrumentation algorithm. $\mathcal{I}$ modifies the definition of sub-agents `breakTheGlass`$^{\texttt{Authorization}}$, `mendTheGlass`$^{\texttt{Authorization}}$ and `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 trigger event) and `mendTheGlass` (negative trigger event).

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. The implemented tool, `LogInst.v2` [16], is an extension of `LogInst` [17] described in [12]. Our review of `LogInst.v2` is followed

by a demo on the MRS 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 [12] 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 [18] 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 audit 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`, and `log-db` implement Δ, Σ, and Λ components of the model resp.

`LogInst` relies on AspectJ, in particular *before* aspects, to extend the functionalities of trigger and logging event microservices for audit 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 accumulates 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 microservice 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 II 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.

B. *LogInst.v2*

Structurally, `LogInst.v2` instruments microservices similar to `LogInst`, i.e., it adds repositories, REST controllers, web clients and API to communicate with Prolog in the same style as the earlier version. 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, the event is stored in that repository. Moreover, each negative trigger microservice is extended with a REST controller, similar to positive trigger microservices, that responds with the content of the `local-db` repository.

`LogInst.v2` differs from `LogInst` mainly in dealing with logging specifications. 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 redact the negative triggers and build 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 events to `local-db`, and reach out to trigger microservices to store positive and negative 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. A_0 and B_0 are the only constant arguments of this query. The resolution of the query is sent by the Prolog engine and is temporarily stored in a container by the logging event microservice. Let's call this data structure `lg-list`. This list is the collection of all LG predicates, where each predicate refers to a candidate event that could be potentially logged. Next, the microservice checks whether $\mathrm{LoggedCall}(A_0, B_0, xs_0)$ 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, i.e., `LogInst.v2` adds m clauses of the form given in Figure 6 to the engine.

The logging event microservice goes through each candidate in `lg-list` and checks whether $\mathrm{NegativeTrigger}$ predicates are derivable. The algorithm in Figure 7 describes logging event's behavior to infer whether $\mathrm{LoggedCall}(A_0, B_0, xs_0)$ is derivable. Note that `LogInst.v2` does not hard code any of

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

Fig. 6: The clause for the jth negative trigger.

```
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.

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.

Figure 8 summarizes the communication differences between the instrumented logging event microservice and the Prolog engine in `LogInst` vs. `LogInst.v2`. While `LogInst`-instrumented logging event microservice communicates the logging specification with the Prolog engine, and queries the engine for any potential logging, `LogInst.v2`-instrumnted version does not do so due to higher expressivity of the specifications. Instead, it creates an intermediary Horn clause as well as a set of negative trigger clauses out of the specification and communicates them with the Prolog engine. When a logging event transpires, that microservice queries Prolog engine for all LG predicates, follows the pseudocode in Figure 7, queries NegativeTrigger predicates, and accordingly deduces whether the event must be logged.

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 3. 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 in this own thread and communicates different logical rules and facts with it. 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 9. The reader is referred to [12] for more details on these structural modifications. In the following, we focus on the aspects that differ from `LogInst`'s output.

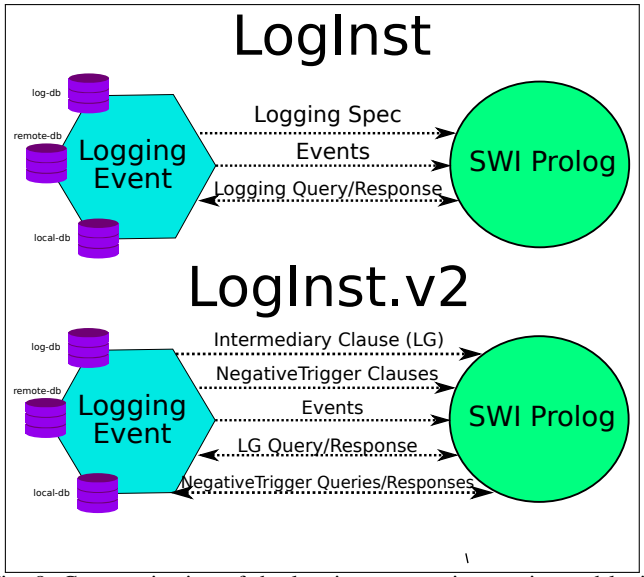

Fig. 8: Communication of the logging event microservice and logic engine: `LogInst` vs. `LogInst.v2`.

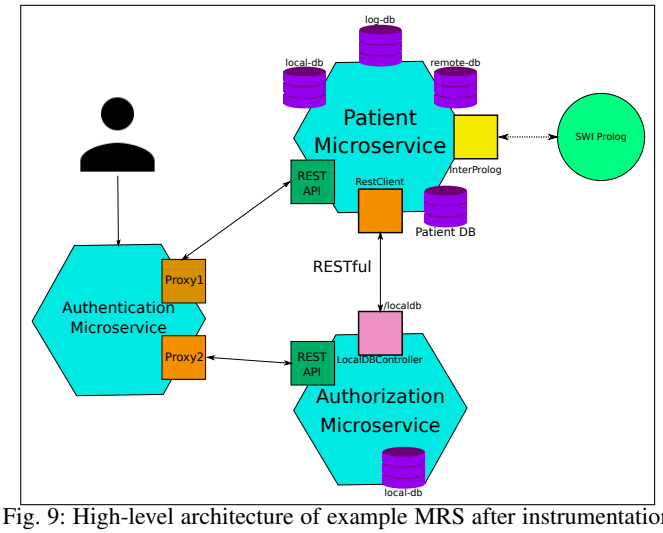

Fig. 9: High-level architecture of example MRS after instrumentation by `LogInst` and `LogInst.v2` using the specification of Figure 3.

```
/* 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).

/* Single Negative Trigger Clause */
negative_trigger(1,T0,T1,[U,P]) :-
  funccall(T2, authorization-service, mendTheGlass, [U]),
  <(T2, T0),
  <(T1, T2).
```

Fig. 10: The intermediary and the negative trigger clauses for the example specification in Figure 3.

`LogInst.v2` modifies Patient service to create an intermediary clause and communicate it with Prolog engine. Moreover, since there is a single negative trigger (`mendTheGlass`) 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. Figure 10 demonstrates the intermediary and negative trigger clauses. Note that in the presentation of these rules, each method's full package/class path is redacted for brevity.

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 results in `lg-list`. Then, Patient service follows the pseudocode in Figure 11 to infer whether logging events must be logged. Note that Figure 11 depicts a specialization of the general process described in 7.

```
for each candidate lg(patient-service, getPatientMedHistory,T0,T1,[U,P])}
          in lg-list:
  if negative_trigger(l,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. 11: Example pseudocode of LoggedCall inference.

IV. Related Work and Conclusion

In this section, we review the related work, and conclude the paper.

A. Related Work

As microservices have gained more popularity in the application design and deployment in recent years, their safety and security have been the focus of several studies, e.g., [19]–[22]. Most commonly in the realm of audit logging, a central approach has been considered, where a specific microservice is responsible to collect all logging events [23], [24]. In particular, Elascale [25] is a monitoring system that is deployed as an independent microservice. In contrast, Amir-Mohammadian et al. [12] propose an instrumentation technique that enables concurrent audit logging in different microservices. `LogInst` [17] is the implementation of this instrumentation technique in the context of Java Spring microservices. Formal correctness of `LogInst` relies on information algebraic [5] semantics of audit logging, originally explored in [9] for concurrent systems.

Operating system-level and network-level monitoring has been a commonplace trend in audit logging for microservices. Examples include Amazon CloudeWatch [26], Nagios [27], Microsoft Azure Kubernetes [28], and Spring Security Framework [29]. Cinque et al. [30] propose a blackbox tracing mechanism for monitoring microservices that does not involve instrumentation.

In this paper, we propose a more powerful tool to support concurrent audit logging in microservices, called `LogInst.v2`. The formal correctness of our work relies on an implementation model on concurrent systems [13].

B. Conclusion

In this paper, we have proposed an instrumentation tool for Java Spring based microservices to enforce audit requirements, whose logical specifications are not expressible by Horn clause logic. Our implementation tool is an extension of an earlier solution that supported specifications in Horn clauses. Our tool receives the specification in JSON format, along with the source code of microservices. It identifies the triggers and logging events in the application, and accordingly instruments them. As a case study, we discuss a microservices-based medical records system that lets users to deactivate controlling access to patient medical information in critical scenarios and activate it at a later stage. The correctness of the instrumentation tool relies on a formal model of the instrumentation that guarantees the generated logs to be necessary and sufficient.

References

[1] "CWE-778: Insufficient Logging," https://rb.gy/2hhb5o, 2021, accessed: 2022-04-04.

[2] "Top 10-2017 A10-Insufficient Logging & Monitoring," https://rb.gy/mj1xpf, 2017, accessed: 2022-04-04.

[3] "CWE-779: Logging of Excessive Data," https://rb.gy/myvjgc, 2021, accessed: 2022-04-04.

[4] "Why Do Attackers Target Industrial Control Systems?" https://rb.gy/ketznl, 2017, accessed: 2022-03-29.

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

[6] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Correct audit logging: Theory and practice," in *International Conference on Principles of Security and Trust (POST)*, April 2016.

[7] S. Amir-Mohammadian and C. Skalka, "In-depth enforcement of dynamic integrity taint analysis," in *PLAS*, 2016.

[8] C. Skalka, S. Amir-Mohammadian, and S. Clark, "Maybe tainted data: Theory and a case study," *J. Comput. Secur.*, vol. 28, no. 3, pp. 295–335, April 2020.

[9] S. Amir-Mohammadian and C. Kari, "Correct audit logging in concurrent systems," *ENTCS*, vol. 351, pp. 115–141, September 2020.

[10] "O'Reilly's Microservices Adoption in 2020 Report Finds that 92% of Organizations are Experiencing Success with Microservices," https://rb.gy/frajim, 2020, accessed: 2022-03-29.

[11] "Organizations' adoption level of microservices worldwide in 2021," https://rb.gy/kanclv, 2022, accessed: 2022-03-29.

[12] S. Amir-Mohammadian and A. Y. Zowj, "Towards concurrent audit logging in microservices," in *Proceedings of the 45th Annual IEEE Computers, Software, and Applications Conference (COMPSAC 2021)*, July 2021, pp. 1357–1362.

[13] S. Amir-Mohammadian, "Correct audit logging in concurrent systems with negative triggers," University of the Pacific, Tech. Rep., June 2021.

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

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

[16] N. D. Ahn and S. Amir-Mohammadian, "LogInst.v2: Instrumenting Java Microservices for Audit Logging: Beyond Horn Clauses," https://rb.gy/vbxtpq, 2022.

[17] S. Amir-Mohammadian and A. Y. Zowj, "LogInst: Instrumenting Microservices of Java Web Apps for Auditing," https://rb.gy/h5gihs, 2020.

[18] "SWI Prolog," https://www.swi-prolog.org/, accessed: 2022-04-03.

[19] N. Mateus-Coelho, M. Cruz-Cunha, and L. G. Ferreira, "Security in microservices architectures," *Procedia Computer Science*, vol. 181, pp. 1225–1236, 2021.

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

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

[22] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of microservices-enabled fog applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 22, p. e4436, 2019.

[23] A. Barabanov and D. Makrushin, "Security audit logging in microservice-based systems: survey of architecture patterns," *arXiv preprint arXiv:2102.09435*, 2021.

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

[25] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: autoscaling and monitoring as a service," *arXiv preprint arXiv:1711.03204*, 2017.

[26] "Amazon CloudWatch," https://rb.gy/da5tsp, 2022, accessed: 2022-04-04.

[27] "Nagios," https://www.nagios.org, 2022, accessed: 2022-04-04.

[28] M. Wasson, "Monitoring a microservices architecture in Azure Kubernetes Service (AKS)," https://rb.gy/xzlm95, 2020, accessed: 2022-04-04.

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

[30] M. Cinque, R. Della Corte, and A. Pecchia, "Microservices monitoring with event logs and black box execution tracing," *IEEE Transactions on Services Computing*, 2019.