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, e.g., for the sake of access control and/or audit logging. Formal correctness of this approach for audit logging relies on semantic frameworks that leverage information algebra to model the information content of the generated audits logs and the program at runtime. Previous work in this realm 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

Insufficient audit logging has been a major application security problem. Proper monitoring of security-critical events in an application can potentially mitigate major data breaches. Open Web Application Security Project has identified insufficient logging and monitoring as one of the top ten security risks in the web [1], and Common Weakness Enumeration system recognizes it as a recurrent problem in software security [2]. In addition, efficiency of audit logging plays a crucial role in system performance. Efficient audit logging entails to only record what is necessary for a posteriori analysis and recovery, rather than naively collecting excessive information in the log that hinders timely response to security incidents [3].

In this regard, an information-algebraic [4] semantic framework has been proposed to define audit logging correctness [5], which ensures to record factual, necessary and sufficient data in the log, and thus avoids both insufficient and excessive logging. This is accomplished by comparing the information contained in the log and the information that *must be* in the log. The latter refers to the specification of audit logging requirements. These specifications recognize what must be logged, given an execution trace. This way, the semantic framework supports the separation of policy from programs, which underlies program instrumentation techniques to implement audit logging on legacy systems.

Different implementation models of correct audit logging have been proposed with provable guarantees. For example, the semantic framework of audit logging is used to define an implementation model for linear process execution [5]. Using this implementation model, for instance, audit logging capability is considered as an extension to a medical records system (MRS), where all preconditions for logging depend on the events that transpire in the same program execution thread.

Recently, an implementation model has been proposed for concurrent systems, where logging an event may be conditioned on the occurrence of events in one or more concurrent components [6]. This model proposes an algorithm to instrument concurrent systems that are specified in a process calculus, and any instrumented system provably guarantees correct audit log generation. The algorithm receives a formal specification of audit logging requirements along with the source concurrent system as input. This specification uses Horn clauses to assert which events should be logged as well as the preconditions to log those events. Using Horn clauses is helpful in actual implementations, since it facilitates to use off-the-shelf logic programming tools.

In this paper, we discuss one such implementation of the instrumentation algorithm for concurrent systems, based on the aforementioned model. Our tool receives the source code of a microservices-based application as input, along with a specification of logging requirements in JSON format. The application is assumed to be deployed in Java Spring framework. The tool parses the JSON specification of requirements and translates them to Horn clauses that are supplied to a logic programming engine. It then instruments the application with audit logging capabilities. The instrumented application communicates with the logic programming engine in appropriate places to infer what to log.

In recent years, there has been a growing trend toward the deployment of applications with microservices architecture. For example, microservices-based healthcare is anticipated to experience considerable increase in market value in near future [7]. In this architecture, the system is decomposed into a set of loosely-coupled, minimal and fine-grained processes that are

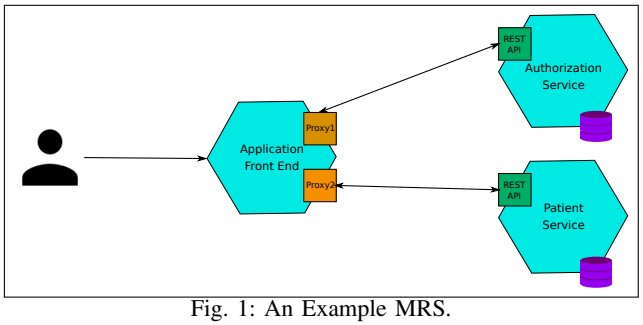


Fig. 1: An Example MRS.

executed independently and collaboratively. Each microservice has its own back-end database and can be executed in its own container or machine. Communication between microservices is usually done through message passing, in particular using RESTful APIs. Microservices come with several advantages including better maintainability, testing, and adaptation to newer technologies, and thus improved security and fault tolerance. In today's world, microservice deployment is supported by several conventional programming languages.

Example: Microservices-based MRS. As an example, consider an MRS with microservices architecture. An MRS may include different microservices to accomplish different tasks. Figure 1 depicts an oversimplified system consisting of a front-end service, an authorization service, and a patient service among other services. Application front-end includes API gateway that relays requests from clients to back-end services, using certain proxies. Patient service handles patient data, and authorization service manages different operations related to controlling access to system resources. One such operation is breaking the glass [8]. This operation is used in critical cases to bypass access control. Breaking the glass requires the users who run this operation to be accountable for their actions, i.e., certain actions of the users who bypass access control need be recorded in the log for postfacto accountability analyses. One such action could be reading patient medical history. Indeed, this simple example demonstrates an audit logging requirement where the event to be logged in a given component of the concurrent system is conditioned on a preceding event that belongs to another component. The former event refers to reading patient medical history in patient microservice, and the latter event refers to breaking the glass in authorization microservice. We will return to this example throughout the paper, as we explore the model and its deployment.

Paper outline. The rest of the paper is organized as follows. In Section II, we review the formal process-algebraic implementation model. 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. Some 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 a 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 $\mathcal{LS}_{call}$ to denote this class of specifications. In $\mathcal{LS}_{call}$, each event is modeled as a sub-agent invocation, i.e., a module within one of the agents of the system. Figure 2 depicts the structure of specifications in $\mathcal{LS}_{call}$. $\mathrm{Call}(t, A, B, mathitxs)$ asserts the event of invoking sub-agent B^A at time t with list of parameters xs. ϕ and ψ_j are 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 **??**, 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 **??**). Moreover, Figure **??** describes the logging specification in $\mathcal{LS}_{call}$ that is associated with the break-the-glass policy. In this clause, t_0 and t_1 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.

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

Fig. 2: $\mathcal{LS}_{call}$ clause structure.

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

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

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

Fig. 3: Example logging specification for an MRS.

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 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.

D. Instrumentation Algorithm

Instrumentation algorithm $\mathcal{I}$ takes a Π system and a logging specification from $\mathcal{LS}_{call}$, and produces a $\Pi_{\log}$ system with the following details. $\mathcal{I}$ adds fresh links c_{ij} between every logging event agent A_i and trigger agent A_j, in order to communicate logging preconditions (by sendPrecond and addPrecond prefixes). If sub-agent B^A is a trigger, then its execution must be preceded by callEvent prefix, so that the logging precondition is stored in $\Delta(A)$. If sub-agent B^A is a logging event agent, the execution of B^A must be preceded by callEvent, similar to the previous case. Next, it must communicate on appropriate links (c_{ij}s) with all trigger agents. To this end, B^A is supposed to notify each of those agents to send their collected preconditions, and then it must add them to $\Sigma(A)$. This is done using addPrecond prefixes. Then, it should study whether the invocation needs to be logged, before following normal execution. This is facilitated by emit prefix.

If A_j is a trigger agent then it must be able to handle incoming requests for collected preconditions. This is done by adding a fresh sub-agent to A_j that always listens for requests on the dedicated link (c_{ij}) between itself and the logging event agent. This sub-agent is denoted by D_{ij}. Upon receiving such a request, D_{ij} sends back the preconditions, handled by prefix sendPrecond, and then continues to listen on c_{ij}.

As an example consider the instrumentation of the MRS described in Figures 1 and 3. According to the logging specification, $\texttt{getMedicalHistory}^{\texttt{Patient}}$ is the logging event,

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

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

$\mathcal{C}'_{\mathcal{L}}(\texttt{Authorization})(D_{\texttt{PA}}) = [D_{\texttt{PA}}^{\texttt{Authorization}}(c_{\texttt{PA}}) \triangleq c_{\texttt{PA}}.\mathsf{sendPrecond}(c_{\texttt{PA}}, \texttt{Authorization}).D_{\texttt{PA}}^{\texttt{Authorization}}(c_{\texttt{PA}})]$

Fig. 4: Example Instrumentation of the MRS.

and $\texttt{breakTheGlass}^{\texttt{Authorization}}$ is the only trigger. $\mathcal{I}$ instruments the system as described in Figure 4. A new link $c_{\texttt{PA}}$ is established between the two agents, and sub-agents are instrumented accordingly. In addition, sub-agent $D_{\texttt{PA}}$ is added to `Authorization` microservice that indefinitely responds to the requests from `Patient` microservice on $c_{\texttt{PA}}$.

III. Instrumenting Microservices

In this section, we explain the implementation of the instrumentation algorithm for microservices-based applications, as well as a demo of how the tool modifies these applications based on different logging specifications. In our instrumentation tool, each microservice of an application is treated as an agent of the concurrent system, whereas each method defined in some library of a microservice is considered as a sub-agent.

A. Instrumentation Tool: `LogInst`

We have implemented the proposed algorithm $\mathcal{I}$ for microservices-based applications that are deployed using Java Spring Framework. Our instrumentation tool [**?**], `LogInst`, receives a logging specification along with an application consisting of two or more microservices, and rewrites those microservices accordingly. `LogInst` extends microservices with required RESTful APIs that facilitate the communication between microservices for the sake of audit logging.

The logical specification of logging requirements (Section II-B) is passed as an argument to `LogInst` in JSON format. `LogInst` parses this JSON file and extracts logging specification in the form of Horn clauses, along with identifying triggers and logging events. The paths to different microservices of the application are also passed to `LogInst`. `LogInst` applies modifications to each microservice component according to the logging specification. In addtion, the path to the Prolog engine must be fed to `LogInst`. `LogInst` uses SWI Prolog [10] to logically infer the derivation of logging events according to the logging specification, the set of facts regarding trigger events, etc.

`LogInst` uses aspect-oriented programming (AOP), in particular AspectJ, to weave concurrent logging capability into microservices. For this purpose, `LogInst` extends the Project Object Model of the microservices which need to be instrumented by `spring-boot-starter-aop` dependency.

According to the implementation model, the configuration of the concurrent system includes three different structures to store the logging preconditions that are transpired locally (Δ), logging preconditions that are originated remotely (Σ), and the audit logs recorded by an agent (Λ). These structures are added by `LogInst` as repositories of logical facts that are kept on

nonvolatile memory. If a microservice is only a trigger, then a repository is added to that microservice to store logging preconditions that take place locally in that microservice. However, if a microservice is a logging event, then that microservices is extended with all three types of repositories. We call these repositories `local-db`, `remote-db`, and `log-db`, resp. Note that according to the definition of $\mathcal{I}$ (Section II-D), a trigger is only concerned with locally transpired preconditions (through callEvent prefixes), whereas a logging event needs to access all three types of aforementioned structures (through callEvent, addPrecond, and emit prefixes).

$\mathcal{I}$ extends every trigger with sub-agents, D_{ij}, that send back the locally generated preconditions upon receiving a request on a dedicated link (using sendPrecond prefix). `LogInst` implements this feature by extending each trigger microservice with a REST controller that sends back the content of `local-db`, if it receives an HTTP GET request on the dedicated path `/localdb`. This controller is denoted by `LocalDBController`, henceforth. On the other side, a logging event microservice is supposed to contact the trigger on dedicated links to receive the preconditions that are transpired in trigger agents. `LogInst` facilitates this by extending every logging event microservice with a web client that sends asynchronous HTTP GET requests to `/localdb` path on trigger microservices and collects the responses. We have called this service `RestClient`.

As mentioned earlier, AOP is used to add logging capabilities to microservices. For this purpose, `LogInst` identifies the pointcuts in which an advice needs to be defined. According to $\mathcal{I}$, trigger sub-agents are preceded by callEvent prefixes. For this purpose, `LogInst` defines *before* aspects for each of the trigger methods, where the advice includes the construction of preconditions from the join points and adding them to `local-db`. This implements the semantics of callEvent.

$\mathcal{I}$ instruments logging event sub-agents by inserting a sequence of operations before the execution of these sub-agents. This sequence includes callEvent prefixes, communication on dedicated links to receive remotely transpired logging preconditions, adding them to Σ using addPrecond prefixes, and finally checking if logging events should to be logged, using emit. `LogInst` handles this by defining *before* aspects for the pointcuts that correspond to logging event methods. Similar to the aspects defined for trigger methods, the advice for logging event methods starts with the construction of preconditions from join points and adding them to `local-db`. Next, `RestClient` is used to asynchronously send an HTTP GET request on the predefined path `/localdb` to each of the trigger microservices, and collect the results in `remote-db`. This implements the semantics of addPrecond prefix. Then, SWI Prolog engine is invoked to add the logging specification, and the contents of `local-db` and `remote-db`. Finally, the Prolog engine is queried to study if the invocation of logging event must be logged, and accordingly `log-db` repository is updated. These final steps implement the semantics of emit prefix. In order to facilitate the communication between SWI Prolog engine and Java Virtual Machine, InterProlog Java/Prolog SDK [11] is used.

```
@Before("execution (some trigger)")
public void someAspect(JoinPoint){
        ...
        build precondition from JoinPoint
        add precondition to local-db
        ...
}

@Before("execution (some logging event)")
public void someAspect(JoinPoint) {
        ...
        build precondition from JoinPoint
        add precondition to local-db
        ...
        send GET to triggers
        collect the responses in remote-db
        ...
        add logging specification to the engine
        add the content of local-db to the engine
        add the content of remote-db to the engine
        query the engine and update log-db
        ...
}
```

Fig. 5: Pseudocode of *before* advice for triggers and logging events.

Figure 5 specifies the advice for triggers and logging events in general form. Figure 7a depicts some of the aforementioned modifications that `LogInst` applies architecturally to the logging event and trigger microservices.

B. Case Study: Instrumenting MRS_{Demo} with `LogInst`

In Section I, we discussed an oversimplified MRS consisting of several loosely-coupled microservices. We have implemented [**?**] a demo of this system, MRS_{Demo}, consisting of several microservices, using Java Spring Boot [12]. The front-end microservice of MRS_{Demo} authenticates users and acts as the API gateway by relaying requests to the back-end microservices.

In the following, we explain how MRS_{Demo} is instrumented by `LogInst` for a given logging specification. In Figure 3, we have described a logging specification that enforces logging access to patient medical history at any point after breaking the glass. We can assert a similar logging specification rule in JSON [**?**] , which is more verbose than its logical equivalent. `LogInst` parses that JSON specification and constructs the Horn clause given in Figure 6 (ver. 1), which is then added to SWI Prolog engine fact base. Note that in this Horn clause presentation, we have redacted the full package names of the trigger and logging event methods and replaced them with `<package>`, for the sake of space economy. `LogInst` instruments MRS_{Demo} according to this logging specification rule as follows: `spring-boot-starter-aop` dependency is added to the POM of Patient and Authorization services. Authorization service is extended with `local-db`. Patient service is extended with `local-db`, as well as `remote-db`, and `log-db`. Authorization service is extended with the REST controller `LocalDBController` that responds to requests on path `/localdb`. Patient service is extended with `RestClient` web client. A *before* aspect is added to Authorization service with `AuthorizationController.breakTheGlass` as its pointcut. This aspect builds preconditions from the join point and appends them to `local-db`. A *before* aspect is added to Patient service with pointcut `PatientController.getPatientMedHistByName`, to 1) build preconditions from the join point and append

```
/* Version 1 */
loggedfunccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]) :-
 funccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]),
 funccall(T1, authorization-service,
  "<package>.AuthorizationController.breakTheGlass", [U]),
 <(T1, T0), ==(U, user).

/* Version 2 */
loggedfunccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]) :-
 funccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]),
 funccall(T1, authorization-service,
  "<package>.AuthorizationController.breakTheGlass", [U]),
 funccall(T2, authentication-service,
  "<package>.AuthenticationService.authenticate", [U]),
 <(T1, T0), <(T2, T1), ==(U, user).

/* Version 3 */
loggedfunccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]) :-
 funccall(T0, patient-service,
  "<package>.PatientController.getPatientMedHistByName", [U, P]),
 funccall(T1, authorization-service,
  "<package>.AuthorizationController.breakTheGlass", [U]),
 funccall(T2, patient-service,
  "<package>.PatientController.getAllPatients", [U]),
 funccall(T3, authorization-service,
  "<package>.AuthorizationController.getBTGUsers", []),
  <(T1, T0), <(T2, T0), <(T3, T0), ==(U, user).
```

Fig. 6: Different versions of break-the-glass policy specified as a Horn clause.

them to `local-db`, 2) send HTTP GET request on path `/localdb` to Authorization service, and store the results in `remote-db` repository, 3) add the logging specification, and contents of `local-db` and `remote-db` repositories to the SWI Prolog engine, and 4) send queries to the Prolog engine to check derivability of `loggedfunccall` predicates and accordingly update `log-db` with the engine's response.

These changes describe the real-world instrumentation of the MRS, formally given in Figure 4. Note that Authentication microservice is unaffected when instrumented by `LogInst`, as it does not include any trigger or logging event methods according to the logging specification.

The two other versions (Figure 6) are example extensions to the policy ver. 1 . In ver. 2, authenication is considered as an additional trigger. Therefore, in addition to the aforementioned changes, `LogInst` extends Authentication microservice with `local-db` repository, `LocalDBController`, and a *before* aspect (trigger version). The *before* aspect of Patient microservice is also extended with sending HTTP GET requests to Authentication microservice on path `/localdb`, and storing the results in `remote-db`. In ver. 3, two additional triggers are considered in Authorization and Patient microservices. `LogInst` applies the same changes given above, along with defining *before* aspects for each extra trigger. Figures 7b and 7c visually describe some of the aforementioned changes to $\texttt{MRS}_{\texttt{Demo}}$ by `LogInst`, considering each version of the policy.These instrumented versions are accessible in [**?**], along with other examples of logging specifications, and their associated instrumented counterparts.

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.

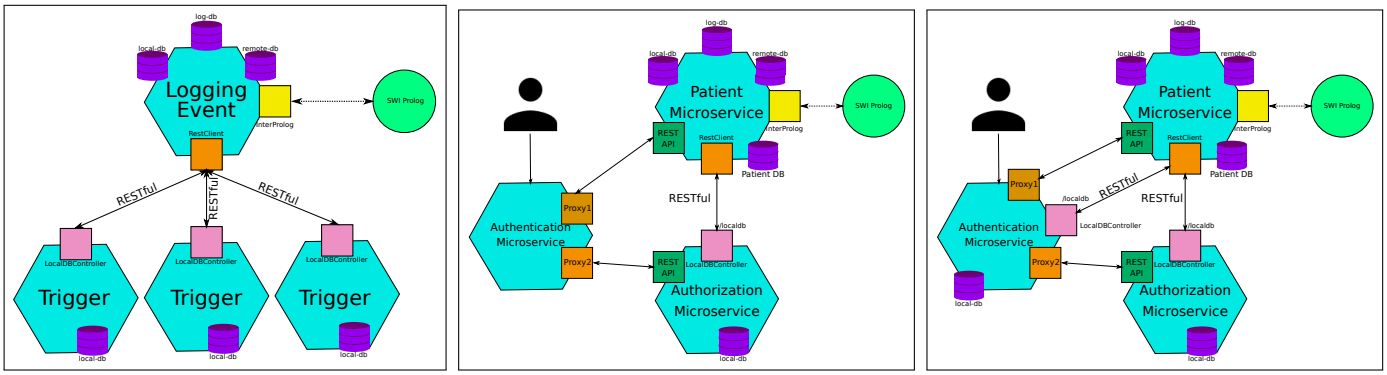
Another line of work uses logical frameworks to establish accountability in access to system resources. Examples include a framework to enforce accountability goals in discretionary access control [31], accountability wrt access to personal information based on owner-defined usage policies [32], distributed 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



(a) Structure of the logging event and trigger microservices, instrumented by `LogInst`.

(b) $\texttt{MRS}_{\texttt{Demo}}$ instrumented by `LogInst` using versions 1 and 3 in Figure 6.

(c) $\texttt{MRS}_{\texttt{Demo}}$ instrumented by `LogInst` using version 2 in Figure 6.

Fig. 7: Architecture of microservices after instrumentation.

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.