Towards Concurrent Audit Logging in Microservices

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

Afsoon Yousefi Zowj  
Dept. of Computer Science  
University of the Pacific  
Stockton, CA, USA  
ayousefizowj@pacific.edu

Abstract—Information-algebraic models have been shown as effective semantic frameworks in the specification of audit logging requirements. These semantic frameworks underlie implementation models that formally guarantee correctness of audit logs. Recently, one such implementation model has been proposed for concurrent systems. In this paper, we study the deployment of an instrumentation tool based on this implementation model, aiming at microservices-based applications that are built by Java Spring framework. This tool instruments these applications according to a given logging specification, described in JSON. A set of events in one or more microservices may necessitate the generation of log entries in a specific microservice. Instrumentation of an application enables different microservices of that application to concurrently generate audit logs.

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 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 toler-
ance. 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. IMPLEMENTATION MODEL

Our instrumentation tool follows a concurrent audit logging model with correctness guarantees. In this section, we briefly review this model. The reader is referred to [6] for the full formalization and correctness results of this model.

A. Source System Model

The source system calculus, denoted by $\Pi$, relies on a variant of $\pi$-calculus [9] to specify concurrent systems. $\Pi$ includes top-level agents $A$ that correspond to the concurrent components of the system, and sub-agents $B$ of top-level agents $A$, denoted by $B^A$. Top-level agents are supposed to execute in parallel and occasionally communicate with each other to accomplish their own tasks, and in aggregate the concurrent system. Sub-agents are treated as internal modules, procedures, or functions of a top-level agent. The existence of universal codebase $C_U$ and local codebase $C_L$ are assumed that respectively consist of top-level agent definitions of the form $A(x_1, \ldots, x_n) \triangleq P$, and sub-agent definitions of the form $B^A(x_1, \ldots, x_n) \triangleq P$, where $P$ is a $\pi$-calculus process.

B. A Class of Logging Specifications

Horn clause logic is used to specify audit logging requirements in this implementation model. In particular, a class of specifications is considered that reflects on time-based relations among different events of a concurrent system, meaning that a particular event must be logged as long as certain set of events, in potentially different components of the concurrent system, have already taken place. These events may need to satisfy certain properties, e.g., following certain ordering in time, etc. This class of specifications is denoted by $LS_{call}$, in which each event is an invocation of an agent module, i.e., a sub-agent. $LS_{call}$ includes a set of Horn clauses of the form given in Figure 2, where $Call(t, A, B, \bar{x})$ specifies the event of invoking sub-agent $B^A$ at time $t$ with list of parameters $\bar{x}$. $\varphi(t_0, \ldots, t_n)$ is a possibly empty conjunctive sequence of literals of the form $t_i < t_j$. $A_0$ is called a logging event agent, whereas other $A_i$s are called trigger agents. Similarly, logging event sub-agent refers to $B_0$, and other $B_i$s are called trigger sub-agents. Logging preconditions are predicates $Call(t_i, A_i, B_i, \bar{x})$ for all $i \in \{1, \ldots, n\}$.

As an example, consider the microservices-based MRS, described in Figure 1. Each microservice is an agent in $\Pi$, including Authorization and Patient services. These agents may respectively include modules to break the glass and read patient medical history, which are defined as part of $C_L$ (Figure 3). Moreover, Figure 3 describes the logging specification in $LS_{call}$ 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.

C. Target System Model

The target system model, denoted by $\Pi_{log}$, is an extension to $\Pi$. The instrumentation algorithm maps a $\Pi$-sys-
tem to a $\Pi_{\log}$ system. $\Pi_{\log}$ is extended with four prefixes: callEvent($A, B, \bar{x}$), addPrecond($x, A$), sendPrecond($x, A$) and emit($A, B, \bar{x}$), in which $\bar{x}$ is considered as a single list of names. The runtime environment is also extended with four components: 1) timing counter $t$, 2) mapping $\Delta(\cdot)$ from agent $A$ to the set of logging preconditions that denote the events transpired locally in $A$, 3) mapping $\Sigma(\cdot)$ from logging event $A$ to the set of all logical preconditions that have taken place in $A$’s trigger agents (these preconditions must be communicated between $A$ and all trigger agents), and 4) mapping $\Lambda(\cdot)$ from $A$ to the audit log recorded by $A$.

callEvent($A, B, \bar{x}$) updates $\Delta(A)$ with predicate Call($t, A, B, \bar{x}$). addPrecond($x, A$) updates $\Sigma(A)$ with precondition $x$. sendPrecond($x, A$) converts $\Delta(A)$ to a transferable object and sends it though link $x$. Finally, emit($A, B, \bar{x}$) studies the derivability of LoggedCall($A, B, \bar{x}$) and accordingly $\Lambda(A)$ is updated with this predicate.

D. Instrumentation Algorithm

Instrumentation algorithm $I$ takes a $\Pi$ system and a logging specification from $\mathcal{LS}_{\text{call}}$, and produces a $\Pi_{\log}$ system with the following details. $I$ adds fresh links $c_{ij}$ between every logging event agent $A_j$ 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, getMedicalHistory$^{\text{Patient}}$ is the logging event, and breakTheGlass$^{\text{Authorization}}$ is the only trigger. $I$ instruments the system as described in Figure 4. A new link $c_{PA}$ is established between the two agents, and sub-agents are instrumented accordingly. In addition, sub-agent $D_{PA}$ is added to Authorization microservice that indefinitely responds to the requests from Patient microservice on $c_{PA}$.

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 $I$ for microservices-based applications that are deployed using Java Spring Framework. Our instrumentation tool [10], 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 parsed 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 addition, the path to the Prolog engine must be fed to LogInst. LogInst uses SWI Prolog [11] 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 ($\Delta$), logging preconditions that are originated remotely ($\Sigma$), and the audit logs recorded by an agent ($\Lambda$). 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 microservice 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 $I$ (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 \( \Sigma \) using addPrecond prefixes, and finally checking if logging events should 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 [12] is used.

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\textsubscript{Demo} with LogInst

In Section I, we discussed an oversimplified MRS consisting of several loosely-coupled microservices. We have implemented [13] a demo of this system, MRS\textsubscript{Demo}, consisting of several microservices, using Java Spring Boot [14]. The front-end microservice of MRS\textsubscript{Demo} authenticates users and acts as the API gateway by relaying requests to the back-end microservices.

In the following, we explain how MRS\textsubscript{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 [10], 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\textsubscript{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 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 loggedfuncall 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.
Formal study of audit logging. One line of work has focused on the security of logs, in particular through cryptographic techniques, e.g., to establish forward secrecy [29], to ensure trustworthiness of logs [30], [31], and to preserve privacy in auditing [32]. 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 [33], accountability wrt access to personal information based on owner-defined usage policies [34], distributed accountability based on turn-based games [35], and logging the proof of having access to system resources [36], [37]. Another related area of work is the language-level analysis of generated audit logs [38], [39].

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 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 [40]–[42]. Recent works in this area include ClearScope [43] a provenance tracker for Android devices, CamFlow [44] an auditing and provenance capture utility in Linux, and AccessProv [45] 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