Audit Trails in the Aeolus Distributed Security Platform

by

Victoria Popic

B.S., C.S. and Math MIT (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September, 2010

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2010

Certified by .....
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Audit Trails in the Aeolus Distributed Security Platform

by

Victoria Popic

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis provides a complete design and implementation of audit trail collection and storage for Aeolus, a distributed security platform based on information flow control. An information flow control system regulates all activities that concern information security. By recording all the operations monitored by Aeolus, our audit trails capture all actions that can affect system security. In our system, event records are collected on each system node and shipped to a centralized location, where they are stored and processed. To correlate audit trail events of different system nodes we store event dependencies directly in the event records. Each audit trail record keeps links to its immediate predecessors. Therefore, our audit trails form dependency graphs that capture the causal relationship among system events. These graphs can be used to reconstruct the chains of events leading to a given system state. Our results show that audit trail collection imposes a small overhead on system performance.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgments

Foremost, I would like to thank my advisor, Prof. Barbara Liskov. Her help on this project was invaluable. I feel very fortunate to have worked with her and learned from her during this year. I'm very grateful for all the time, patience, and effort that she invested in this work. Her vast knowledge and dedication are truly inspiring. I would also like to thank Prof. Liuba Shrira for her useful insights during the course of this project.

A thank you to everyone in 32-G908 for making my time in the lab more enjoyable. Many thanks especially go to James for his friendship and innumerable advices. Also thank you to Aaron for enduring the long discussions on all kinds of platform details. I'm confident that I'm leaving this project in very good hands.

I wouldn't be where I am now if it wasn't for my family. I thank my parents, my grandparents, and my sister for all their care and support throughout the years: thank you for always having trust in me, for the understanding, and for the worries, sometimes about nothing.

I'm also especially grateful to my boyfriend, Zviad, for being so patient and always knowing how to cheer me up.

Finally, thank you, MIT! Thanks to Anne Hunter, my advisors, professors and friends, to the infinite corridor, the coffee shops, 32-123, and my favorite Green building.

Contents

Chapter 1

Introduction

The purpose of this thesis is to add audit trails to Aeolus, a distributed security platform based on information flow control. Aeolus is designed to facilitate the development of secure applications that protect confidential information entrusted to them.

However, even applications implemented in Aeolus can contain errors that lead to violations of security, and furthermore, malicious attacks can cause applications to misbehave. Therefore, it is also important to evaluate what the system does in order to determine whether it is behaving as expected, and if not, to discover what led up to the problem.

Monitoring events that occur in a given system can be achieved through the use of audit trails. Audit trails can be used to detect security violations, diagnose performance problems, and discover programming errors. In this project, we focus primarily on the goal of detecting security violations and assume that the system is operating correctly and performs adequately under normal workload.

Support for auditing requires both collecting the data and providing means to analyze it. This work is primarily concerned with the former. We provide a complete design and implementation for collecting and storing audit trails for applications that run on the Aeolus platform. In addition, we provide tools for examining and displaying the collected information.

1.1 Related Work

This section discusses the relationship of our work to previous research in the area of secure audit trails.

Our approach follows many common practices established by earlier work. For example, we define a format for our event records that includes standard event information, such as the type of the event, its approximate date and time, and success/failure status [19, 5].

However, our project differs from prior work in a number of ways. First, to the best of our knowledge ours is the first system that does auditing for a platform based on information flow control instead of access control. Information flow control is a reliable way to protect information because it can allow access to the data without also allowing its release. For example, an administrator in a medical system may need to examine patient records, e.g., to set up appointments. With access control, allowing an individual to use information also allows him to release it, e.g., attach it to an email. An information flow control system separates these activities, avoiding information leaks.

A system based on information flow control provides a useful way of determining what events need to be included in the audit trails. In particular, an information flow control system regulates all activities that concern information security. This means that to get a complete audit trail of actions that affect system security, it is sufficient to capture all the operations that are monitored by the system; this is the approach we take in auditing Aeolus.

In this project we focus on auditing in a distributed system. Although auditing has been an active area of research for many years, much of the earlier work deals with stand-alone single-processor machines [14, 18]. Auditing in a distributed setting, which includes multiple processors operating asynchronously, introduces several additional concerns [17]. One problem is to determine where and how to collect the logs efficiently. We address this question by following a common audit trail collection pattern; namely, we collect the logs locally on each node and then ship them to a

central location for storage and processing [2].

A much more fundamental problem is how to correlate events recorded on different nodes of the system. On a single processor, events can be ordered using the machine clock. However, this approach does not work in a multi-processor environment. Although extensive work has been done on distributing a standard time among system nodes (e.g. NTP [12]), the clocks cannot be synchronized precisely. Nevertheless many distributed systems rely on the availability of accurate timestamps for event ordering; a discussion on this topic can be found in [15].

Lamport logical clocks [10] can be used to get a "happened-before" ordering with minimal overhead. These clocks guarantee that if an event happened before another event, then its logical clock value will be smaller than that of the second event. However, the converse is not necessarily true, i.e., if the clock value of an event is smaller than that of a second event, it is not necessarily the case that this event happened before the second event. Therefore, the total ordering provided by the Lamport clocks is one of many possible total orderings of the system events.

In general, ordering events based on time (i.e. the "happened-before" relationship) cannot be relied on to imply a causal relationship since it does not provide information about how the events actually depend on each other, i.e. about what events could have causally affected others. Vector timestamps [8] derived from Lamport clocks provide a way to capture this information; however, they are not scalable and will cause significant overhead if used in a large distributed system.

In this work we explicitly correlate audit trail records so that we can reconstruct all the dependencies among the recorded events. This is achieved by uniquely identifying each recorded event in the system and storing event dependencies directly in the audit trail. Each audit trail record stores links to its immediate predecessor events. Therefore, our audit trails form dependency graphs, where the nodes are the individual system execution events and the edges represent the causal relationships captured by the predecessor links. From this information, we can easily reconstruct event chains by following the event predecessor links. For example, we can find all the events that could have led to an invalid system state.

Recent work on backtracking intrusions [9] presents dependency graphs constructed by recording system calls that induce dependency relationships between OS-level objects (e.g. creating a process, reading and writing files). These graphs differ from the ones generated by our audit trails since their nodes represent the individual system objects (e.g. files and processes) and the edges represent system calls that involves these objects (e.g. a process executing a file). In our system, as explained above, the nodes are the events recorded during system execution (e.g. file read, I/O device write) and the edges capture their causal relationship (e.g. a file read affects the subsequent write of a different file).

There is a large body of work on audit trail discovery, e.g. intrusion detection. A survey of this work can be found in [11, 15]. This work is concerned with discovering anomalous user behavior, for example, using statistical analysis or attack signature-detection [20, 22, 21, 13]. Although our project is primarily concerned with audit trail collection, we need to ensure that our audit trails capture all the information necessary to enable a useful audit trail analysis. Prior work introduced the concept of goal oriented logging [3, 2]. This is the idea that the goals of the auditing process should determine what information is logged. Bishop *et al.* [3] propose deriving audit trail content requirements from the security policy of the system that partitions all system states into secure and non-secure. The content of an audit trail then needs to monitor all the events that can lead from a secure to a non-secure system state.

Our audit trails monitor events at the level of the user application interaction with the Aeolus security platform. Logging at the interface to the Aeolus platform allows us to capture all the security-sensitive system activity. As mentioned earlier, this is because any operation that uses or affects information flow, which is at the basis of our security model, is mediated by the Aeolus platform. In addition, we also provide an interface to our auditing subsystem that can be used to capture application-level events.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 presents the relevant background on the Aeolus security platform and its implementation. In Chapter 3 we discuss how we reconstruct the history of program executions using Aeolus audit trails. Chapter 4 describes the format of audit trail records and what kind of information they capture. In Chapter 5 we provide specific information about the events stored in the audit trail. Chapter 6 describes the implementation techniques used to build the audit trail collection framework and Chapter 7 presents several examples of how audit trail data can be used to monitor and evaluate system activity. Chapter 8 evaluates the audit trail overhead on the performance of the system. Finally, Chapter 9 concludes and discusses future work for this project. Appendix A provides a complete summary of all the events recorded in the audit trails and Appendix B includes a detailed description of the data structures used to collect and ship audit trail data in our current prototype.

Chapter 2

Aeolus Platform

A complete description of the Aeolus platform can be found in Cheng[6]. In this chapter we briefly present some relevant Aeolus background information. We begin by describing the high-level platform architecture in Section 2.1 and then introduce several key implementation concepts in Section 2.2. Any major differences between the current platform implementation and the work presented in [6] will be discussed in the appropriate sections of this thesis.

2.1 Aeolus Architecture

Aeolus is a distributed security platform based on information flow control: it tracks information as it flows through programs and determines whether programs have the authority to perform certain security-sensitive operations. Figure 2-1 shows the high-level architecture of the platform. This figure shows the Aeolus platform running on a collection of nodes divided into two categories: compute nodes and storage nodes. Compute nodes are intended to run user application code and storage nodes are used to persistently store sensitive user data (note: a node can be both a compute and a storage node). Aeolus also includes a logically centralized shared authority state (AS). Aeolus nodes consult the authority state in order to determine whether privileged operations should be allowed. Communication can occur among Aeolus system nodes, as well as with outside nodes and I/O devices, which are not trusted.

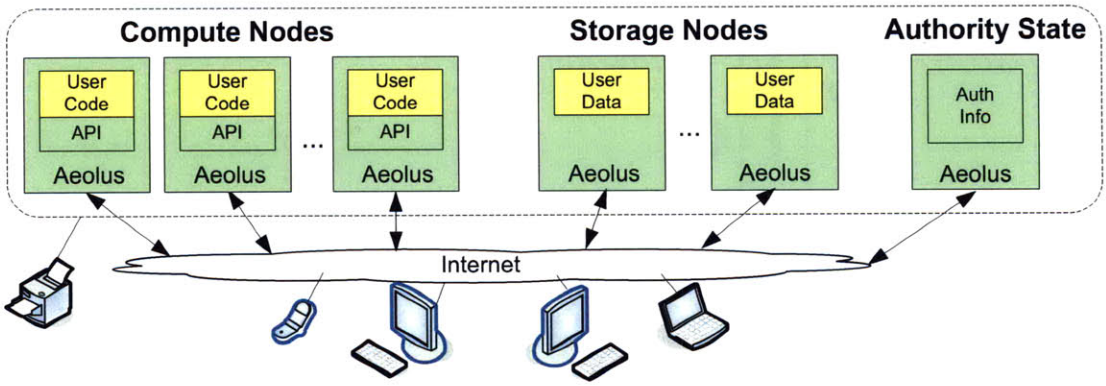

Figure 2-1: Aeolus Distributed Security Platform

More details about the Aeolus platform architecture can be found in Chapter 2 of [6].

2.2 Aeolus Platform Implementation

As shown in Figure 2-2, each compute node runs a set of platform instances (PIs). Within each platform instance there is an Aeolus system process, which has access to all system resources, and many user threads managed by the Aeolus process. User threads make calls to the Aeolus process to get access to various system resources (e.g. the file system and I/O devices). User code is executed in the user threads. It interacts with the Aeolus platform through a set of methods exposed by the Aeolus User API. Chapter 3 of [6] contains a detailed description of all the provided methods. Each platform instance also has an audit trail collector object that is responsible for collecting and shipping audit trail events from all the user threads in the platform instance and a shared state manager that provides user access to shared memory.

In order to allow RPCs among Aeolus nodes, each node maintains a registry of services provided by platform instances at that node. In addition, each node also includes an authority state client that manages interactions with the authority state server and file system clients that manage interactions with the corresponding storage nodes.

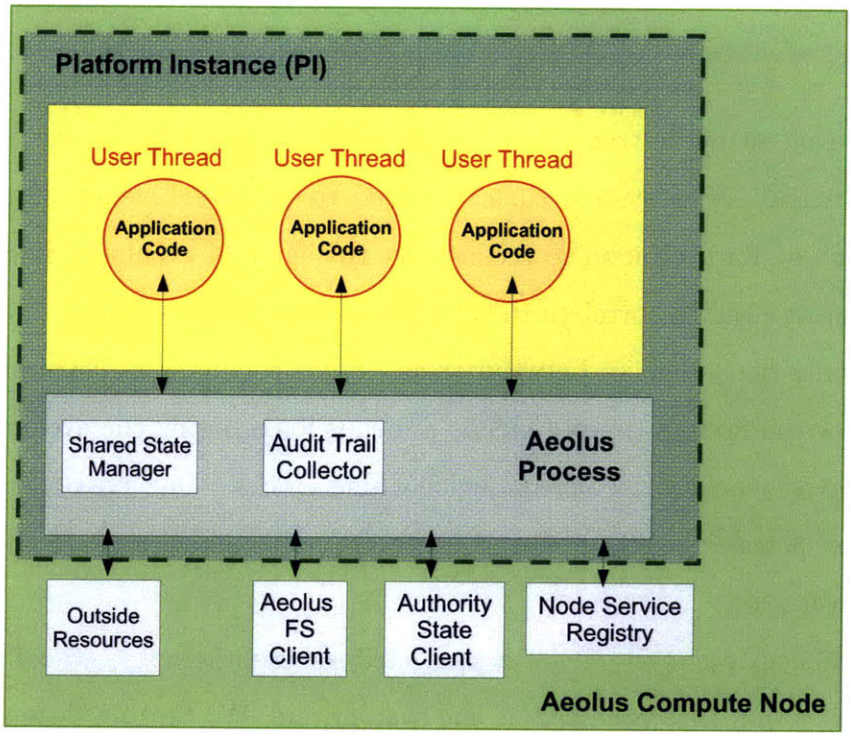

Figure 2-2: Aeolus Compute Node Structure

Chapter 3

Event Graphs

The purpose of an audit trail is to enable the reconstruction of the history of a system execution. As a system runs, we need to capture "events" that represent what it is doing. Furthermore, to reconstruct the chain of events leading to a given system state, we need to correlate the events in the system - we need to order them, recognizing any dependencies between them. Since Aeolus is a distributed system, we cannot rely on system timestamps as a means for ordering the events due to the possible lack of synchronization among machine clocks. One possibility would be to use vector timestamps [8]; however, these are not scalable and would result in a significant overhead.

Audit trails in our system are represented as event graphs, where each event records some action that occurred as the system ran. We capture dependencies between events by storing them directly. Each audit trail event stores links to the events that immediately preceded it. Following these predecessor links, we can reconstruct a partial event order and thus correlate certain events in the log. More specifically, given an event in the log, E, we define $Pred(E)$ to be the set of events that are immediate predecessors of E (we describe what these predecessors are in Chapter 5). Iteratively applying the predecessor relation starting with E:

$$E \rightarrow Pred(E) \rightarrow Pred(Pred(E)) \rightarrow ...,$$

we can order all the events that E depends on and thus find, by transitivity, all the

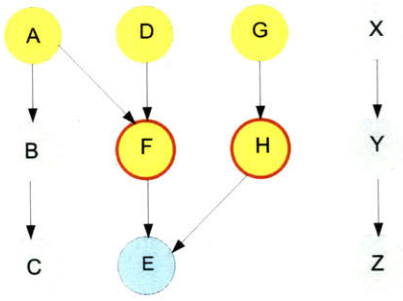

Figure 3-1: Event Dependency Graph. Predecessors of event E are highlighted in yellow; direct predecessors are contoured in red.

events that could have caused E.

In other words, audit trail event entries together with their predecessor links form an *event dependency graph* of all the events recorded during the run of an application. These event dependency graphs are *directed acyclic graphs* (DAGs) where the nodes represent the recorded events and the edges denote the event predecessor relationship (i.e. there will be an edge $A \rightarrow B$ in the graph if node A is the direct predecessor of node B, that is, the set $Pred(B)$ includes A). Therefore, a given event P is a predecessor of another event E if there is a path from node P to E in the event dependency graph and so all the events that could have caused E are the nodes in the graph that have a path to E. These graphs have no cycles due to the temporal nature of the predecessor relation: if an event A is a predecessor of an event B, then it must have occurred prior to B (note: there are no self-edges in these graphs since an event cannot be its own predecessor). In order for a cycle to occur, there needs to be some event C such that B is a predecessor to C and C is a predecessor to A; however, if B is a predecessor to C, then C must have occurred after B and, by transitivity, after A and so it cannot be a predecessor to A.

Figure 3-1 shows a sample event dependency graph. The direct predecessors of E are the events F and H (i.e. $Pred(E) = \{F, H\}$). Events A, D, and G are also E's predecessors since there is a path from these events to E ($Pred(Pred(E)) = \{A, D, G\}$).

It is important to note that predecessor links do not provide us with a total ordering of all the events in the system and only allow us to order events that somehow depend on each other. Concurrent events will be part of separate predecessor dependency chains and cannot be correlated. For example, in Figure 3-1 we cannot correlate the chain of events $X \rightarrow Y \rightarrow Z$ with any other events in the graph. However, since each event entry in the log includes an approximate timestamp, we will be able to recognize the order of events occurring at sufficiently different times.

Chapter 4

Audit Trail Content

An Aeolus audit trail is an event log that monitors user code interaction with the Aeolus platform; more specifically, it records the invocation of every method exposed by the Aeolus User API. Each use of the Aeolus platform corresponds to one or more event records in the log. By logging at the level of the Aeolus interface, we guarantee that we can capture all security-sensitive operations. We also provide an interface to the auditing subsystem that can be used to capture additional events. This interface can be used by an application to record events that correspond to application level activities. The Aeolus audit trail could also include events that correspond to internal Aeolus platform actions. These events would be useful in debugging the platform code; however, we do not investigate this use in this work.

We begin in Section 4.1 by describing the format and content of the recorded events. Section 4.2 provides more details about what are the predecessors of an event and introduces the terminology used to refer to them throughout the thesis. Finally, Section 4.3 describes the operational interface to the auditing layer that can be used to record additional application level events. Chapter 5 describes what events are placed in the log for all the operations that make up the interface to Aeolus.

4.1 Event Record Format

An audit trail event record has the following format:

<eid, preds, method-name, parms, status, ret-val, timestamp>.

We now briefly describe each included field.

Event ID (eid)

Each event record is associated with a globally unique event ID. This ID is used to refer to a particular event from other event records.

Event Predecessors (preds)

Event predecessors are a set of event IDs that correspond to events considered to have immediately preceded the given event; that is, events upon which the given event depends directly. We store the IDs of the predecessors to allow us to correlate and order the event entries in the log. Since each event has a globally unique ID, the stored predecessor IDs will always resolve to the appropriate events. Section 4.2 describes how the predecessors are determined.

Method Name (method-name)

The method name identifies the type of the recorded event (e.g. a file read or a principal creation).

Parameters (parms)

The parameters are the user-supplied arguments to the methods of the Aeolus API that are relevant to Aeolus (e.g. the name of the file a user is trying to read or the labels for a shared state object to be created). We do not log application level user-supplied arguments, such as arguments to remote procedure calls, or the data being written to a file or to a shared state object. This information can be recorded at the application level using the interface for application level logging. Storing event parameters allows us to filter on events involving arguments of interest; for example, we might want to track all the delegations of a given tag or find all the failed attempts to release a sensitive file. Although the current prototype does not

support backtracking, user-supplied parameters can help support this functionality in future work.

Status (`status`)

The status denotes the event outcome: whether it succeeded or failed. An operation can fail in an expected way due to an illegal argument or non-deterministically due to an unpredictable cause, such as a system network or I/O exception. Although we assume the system is operating correctly (and hence will not generate exceptions due to system code bugs), we would not be able to tell whether an operation actually succeeded (and hence induced some changes to the state of the system) due to possible unpredictable failures associated with many Aeolus operations. For example, we can predict any failures due to information flow constraints or access of non-existent files or shared state objects; however, we cannot predict a network failure during an RPC or authority client communication with the authority server. Although it might be useful to diagnose certain problems, we do not explicitly specify the cause of an operation failure since it is not needed to satisfy our primary security goal specified above.

Return Value (`ret-val`)

The return value of an event is an Aeolus-related result generated by the operation. As with parameters, we do not log application-level information, such as the data returned from a file read. The return value can be useful for event log filtering; for example, we might be interested in the creation event of a given principal, tag, or shared state object - this event can be found by matching the return value with the given object ID.

Timestamp (`timestamp`)

The timestamp provides an approximate time at which the event occurred. Although local system timestamps cannot be used to order events in a distributed system, they can be useful for filtering audit trail data; for example, they can be used to answer

questions about what was happening in the system approximately before, at, or after some given time of the day.

4.2 Event Predecessors

Most events are recorded during the execution of a user application. These events always have at least one direct predecessor event, which is the event that immediately preceded them in the same user thread, referred to as P-PRED. More specifically, this event is the latest interaction with the Aeolus platform of the same user computation. The P-PRED links provide a way to order all the events in a given user process.

If there were no communication between user processes, P-PRED links would be sufficient to capture most of the user event dependencies. However, Aeolus provides several mechanisms for inter-process communication: user processes can communicate via the shared state, the file system, remote procedure calls, and, implicitly, through the authority state. Any point of communication between two separate processes creates a dependency between the events in these processes. For example, if a process A writes to a file and another process B later reads from this file observing what A wrote, then all the events in B starting with the file read depend on A's file write event; therefore, the file write event serves as a predecessor to the file read and, by transitivity, to all the subsequent events of B. For this reason, audit trail entries corresponding to communication events usually store a second predecessor, C-PRED, that may represent an event in a different user process.

To summarize, an event can generally have two predecessors: P-PRED and C-PRED. P-PRED is the event that immediately preceded the given operation in the same user process and C-PRED is an event of a possibly different user process that also logically precedes the given operation due to some form of inter-process communication.

The mechanism of determining and assigning C-PREDs is different for each communication mechanism. Chapter 5 describes the mechanisms that Aeolus provides for inter-process communication and what the logical C-PREDs are for each type of

their explicit or implicit communication events.

4.3 Auditing User Interface

Although Aeolus records all security-sensitive operations executed by the user application code, developers might be interested in including additional events into the audit trails. We provide a simple interface to our auditing subsystem, which allows adding application level information to Aeolus audit trails. This interface is an extension to the Aeolus User API.

In order to add an event to the audit trail, applications can use the `CreateEvent` operation. This operation creates and stores a new event in the audit trail and returns the ID of this event. This operation is treated like any other invocation of the Aeolus User API and the new event is correlated accordingly with other events in the same user process; namely, its P-PRED is set to the previous invocation of an Aeolus API method by the same user thread. Applications can use the returned event IDs to correlate their events; namely, they can specify them as predecessors when creating other events. The created event has the same format as all other platform level events; the arguments to the operation correspond to the fields described in Section 4.1.

Creating a new event :

- `eid createEvent(preds, method-name, parms, status, ret-val, timestamp)`: Creates a new event in the audit trail and returns its event ID. The method name must be specified as a string; the parameters and the return value must be serializable objects, so that they can be persistently stored and retrieved. This event automatically has the previous event of the calling thread as its P-PRED; additional predecessors can be specified using the `preds` argument.

Chapter 5

Audit Trail Events

This chapter describes in more detail what event records are included in the audit trail for each type of user code interaction with the Aeolus platform. Each section begins with a brief description of the mechanism involved and then discusses the audit trail content it generates. A summary of events associated with each Aeolus operation can be found in Appendix A.

5.1 Authority State

We differentiate between two types of authority events: updates and lookups. Authority updates (e.g. authority delegations or revocations) are processed at the authority server, while authority lookups (e.g. checks if a principal has authority for a given tag or can switch to another principal) are answered by the local authority client of each Aeolus node.

Although the authority state is not a mechanism for direct user process communication, authority events from one user process can influence the result of authority events that follow them in a different user process and can logically serve as their predecessors. For example, if principal A delegates authority for tag T to a different principal B and the event of checking B's authority for tag T during a declassify succeeds, then the authority delegation event is a logical predecessor of the successful authority check event (assuming B did not already have authority for T from

somewhere else).

5.1.1 Authority Updates

Authority updates can be issued concurrently by different user processes and are serialized at the authority server, which propagates these updates to authority clients. Each update operation corresponds to three events in the audit trail. Two events are logged at the client node. These correspond to the authority update request and reply, respectively. When the authority server receives the update request it logs an additional event that represents the update operation at the server. The C-PRED of this event is the authority update request from the client and its P-PRED is the update event that was serialized immediately before it at the server. The update event at the authority server is the C-PRED of the update reply event at the client; the P-PRED of the reply event is the update request. Figure 5-1 shows the correlation of the events at the authority server and two authority clients.

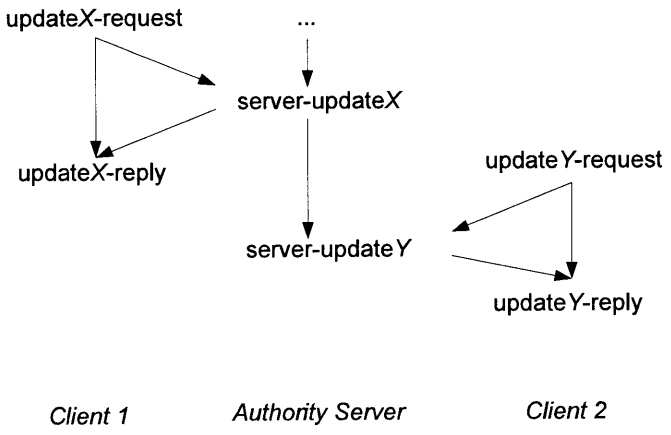

Figure 5-1: Authority Updates Correlation

5.1.2 Authority Lookups

An authority lookup operation corresponds to one event in the audit trail. Authority lookups are processed at the authority client, which caches authority state and may not be up to date with the authority server. The C-PRED of these events is the latest

authority update event seen by their authority client - i.e. the latest authority update reflected in the answer from the authority client.

5.2 Node Creation and Deletion

Once the authority state is running, nodes can be added to the Aeolus system. The creation of an Aeolus node takes place in several steps. First the node is registered with the authority state. As part of node registration, the authority state creates a root principal ID for this node (the first platform instance on that node will run on behalf of this principal ID). After registration, various node components are initialized (for example, the file system manager (if this is a storage node), and the logging client; the node also keeps structures to track platform instances it has launched and the services they provide. Finally, the first platform instance is created on this node to run the start-up application code. When the node is launched, a node creation event is logged. This event record stores information about the node, such as the node's hostname and root principal ID. This event has no P-PRED since it is the first system event on this machine; however, it has a C-PRED event returned as part of the authority state registration reply, representing the registration event at the authority server. After the node creation event, two more events are recorded. First an event is created to represent the launch of the start-up PI. This event has the node creation event as its predecessor; the platform instance launch events are described in more detail in the next section. The second event is the launch of the first user thread in the start-up PI, which has the platform instance launch event as its predecessor and stores the principal ID that the thread runs on behalf of.

A node can also be deleted from the Aeolus system. Upon node deleted, the authority state is notified to remove the node from the system. Two events are created for this operation in the user thread requesting the deletion: the request event, which includes the information about the node (i.e. node hostname), and the reply event. An event is also recorded at the authority state server to represent the deletion at the server.

5.3 Platform Instance Launch and Shutdown

As mentioned in Section 5.2, when a node starts up, Aeolus launches the start-up platform instance. In addition, Aeolus provides a mechanism to launch other platform instances locally and remotely. When a platform instance is created, it is associated with a principal ID (the start-up platform instance of each node runs on behalf of the node root principal). Application developers can specify what principal to launch a platform instance with, as long as information flow rules are obeyed (namely, the caller has null labels and has the authority to act on behalf of the specified principal). The caller also specifies what application to run in this platform instance and any application arguments (the application is specified as the full path name to its class). This code starts running in a user thread of this platform instance.

The launch of a platform instance caused by a call in a user thread corresponds to five event records in the audit trails. The first event is the PI launch request, which includes the parameters discussed above, such as the platform instance's principal ID and the application name (the arguments to the application are not included). This event has only one predecessor; namely, its P-PRED. The second event is logged when the new platform instance is created at the appropriate Aeolus node. This event has as its P-PRED the node creation event and as its C-PRED the PI launch request event; it also stores the the platform instance's principal ID and the application name. The next event is logged when the first user thread of this platform instance is launched to run the specified application. After this application thread finishes, the return from the PI launch event is created. The P-PRED of the return event is the latest invocation of an Aeolus API method in the user application thread. This event becomes the C-PRED of the final event recorded for this operation. This last event is created in the thread that issued the launch request and represents the PI launch reply.

In the case of the start-up platform instance launch only three events are recorded. These events are the events described above excluding the request and reply events of the caller user thread.

Figure 5-2 shows the correlation of the platform instance launch events. The events recorded during the start-up platform instance launch are the three events at the server side; the figure also shows the node creation event that triggers the start-up platform instance launch.

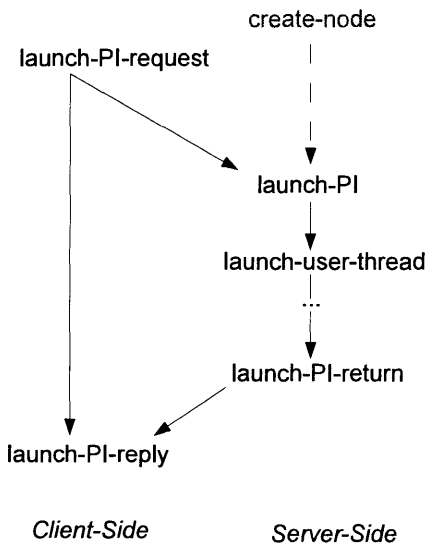

Figure 5-2: Platform Instance Launch Events Correlation

We are interested in knowing what code is being executed at this platform instance. To monitor this, we log the hash value of the loaded application class (this is achieved by extending the default Java ClassLoader to compute the MD5 hash function of all the classes it loads).

A platform instance can also be shut down locally or remotely. This operation corresponds to four events: the shutdown request and reply in the caller user thread and two events for the shutdown at the appropriate node. When shutting down, the platform instance terminates all its user threads and unregisters its services from the node registry. It also ships any remaining event records to the centralized collector (see Chapter 6 for more details about the audit trail collection framework).

5.4 Service Registration

In order to enable communication between system nodes, each node has a registry of services it hosts. A user thread in a PI can register a service with Aeolus that will be handled by that PI and a user thread on a different node can invoke a method on this service (via the RPC mechanism described below). When registering a service, the name of the service and the name of the implementing class are specified. The service registration corresponds to one event in the audit trail. This event stores the name of the service being registered.

5.5 Remote Procedure Calls

Users can invoke methods on remote services registered with Aeolus by retrieving a service proxy from Aeolus at the client side - this proxy serves as a local reference to the remote service object. A method invocation on this proxy triggers Aeolus to send an RPC request from the client node to the node hosting the service. This request carries some Aeolus specific information necessary to start the processing of the RPC request at the server (such as the caller's process labels), as well as information about the invoked service, method, and the user-supplied parameters to the remote method call. When the server receives an RPC request it forwards it to the PI that handles that service; the PI then starts a new user thread with the specified information flow state running the specified service method (this thread runs on behalf of the platform instance principal). When this new thread finishes, it sends back an RPC reply to the client carrying the result of the computation and its process labels in order to propagate any contamination back to the caller.

RPCs cause four events to be added to the audit trail: two events at the client side (send-RPC and receive-RPC-reply) and two events at the server side (receive-RPC and send-RPC-reply). We correlate the events at the client and the server side, such that the first event that occurs at the server side as a result of the RPC request has as its C-PRED the client's send-RPC event and the first event that occurs at the client

after receiving the RPC reply has as its C-PRED the last server-side event recorded during the remote method invocation. This is achieved by carrying predecessor event IDs via the RPC request and reply. Furthermore, the node service registry serves as an indirect communication point for service registration and invocation events. More specifically, if upon the receipt of an RPC request at a given Aeolus node, the requested service is found on that node, then we know that the service registration event is a predecessor to the subsequent events generated by this RPC's receipt. Therefore, a predecessor of the receive-RPC event is the registration event of the service whose method is being invoked.

Figure 5-3 shows the correlation of the client and server side events, as well as the service registration and access events, which is achieved through the platform instance service registry.

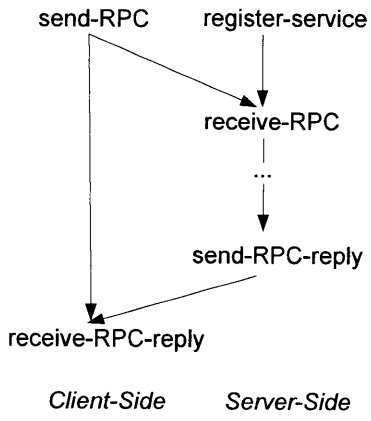

Figure 5-3: Remote Procedure Call Events Correlation

5.5.1 Client-Side

Before a user thread can make remote procedure calls it needs to get the service proxy from Aeolus. This operation corresponds to one event in the audit trail that specifies the name of the obtained service. Each remote method invocation on a service corresponds to two events at the client side. The first event is the send-RPC request, which is generated when a method is invoked on the local service proxy. This

event stores information about the service and the remote method being invoked (the parameters to the remote method are not recorded). The second event is registered upon the receipt of the RPC reply. This event has two predecessors. Its P-PRED is the send-RPC request event and its C-PRED is the last server event.

5.5.2 Server-Side

Similar to the client side events, two events are recorded at the server side for each received RPC request. After a new user thread starts processing the RPC request, the first event in this new thread is the receipt of the RPC request. This event has two predecessors: the send-RPC request event at the client and the registration event of the service being invoked. When the method invocation returns, the send-RPC-reply event is generated, which becomes the predecessor of the receive-RPC-reply event at the client. The P-PRED of this event is the latest event generated by the remote method or, if no events were recorded during the remote method execution, it is the receipt of the RPC request event.

5.6 Local Calls and Forks

Users can make a local call with a different principal by invoking the Aeolus `Call` operation, which runs the user-supplied code object with the desired principal ID in the same user thread. Two events are recorded for this operation: one for the call and one for the return from the call. The call event stores the principal ID that the call runs with; it does not store the actual code. Both events only have one predecessor. The event record for the return event has as its P-PRED the latest event recorded during the execution of the supplied code object.

Users can also fork a new user thread by supplying the code and principal for this thread. An Aeolus `Fork` operation corresponds to two events: a fork request event recorded in the parent thread and the launch of the new user thread event recorded in the child thread. The fork request event is the predecessor of the new user thread launch event. As with the `Call` operation, the fork request stores the principal of the

child thread but does not store the code object being run in that thread.

5.7 Local Authority Closures

Authority closures provide the ability to grant authority directly to code. Users can create authority closures by specifying the full class name of the desired closure to the `CreateClosure` operation. This operation corresponds to one event in the audit trails. A closure creation call is an authority update event and therefore has two predecessors (its C-PRED is the latest authority update event at the server). It stores as a parameter the class name of the closure. It also stores the closure ID as its return value.

To run a closure, users need to get an instance of the given closure type by providing the appropriate closure ID to the `GetClosure` method. This operation also corresponds to one event in the trail. A call to the authority client is made to get the information associated with the given closure ID; this is an authority lookup event and, therefore, this event also has two predecessors (as described in Section 5.1).

Finally, the closure object can be invoked using the `CallClosure` method. This operation executes the closure code inside the same user thread with the authority associated with the closure ID. This is similar to the local call operation. Two events are logged: one for the call and one for the return from the closure.

5.8 Boxes

Boxes encapsulate sensitive data, allowing it to be passed around without contaminating the process unless the box content is accessed. Users can create an `AeolusBox` object by setting its inner and outer labels. Content can be read and written to the box based on the inner labels. A process can also retrieve the inner and outer labels of the box if the box outer labels permit. Each box operation corresponds to one event record. Since users can share data using the box content, the C-PRED of a box content read or write operation is the event that last updated the box content (or

the box creation event, if no content has been written to the box). In this way, we can know what information was observed by a given box content read event and we can capture the order of box content write events. The C-PRED of the inner or outer label retrieval is the box creation event.

5.9 Shared State

The shared state provides a way for user processes of the same platform instance to exchange information and synchronize. There are three ways of communicating via the shared state - through the use of *shared objects*, *shared queues*, and *shared locks*. Each shared state operation corresponds to one event in the audit trail.

5.9.1 Shared Objects

Shared objects allow for content to be stored and retrieved from the shared state by different user processes. Users can create a shared object, read its content, update its content, and delete the object. Each operation (except create) has as its C-PRED the event that *last* updated the content of this shared object. Therefore, a write to a shared state object is a predecessor to any subsequent reads and the subsequent write or delete of this object. The create event is treated as a first write of the object.

5.9.2 Shared Queues

Shared queues provide messaging capabilities to user threads by allowing them to enqueue and dequeue items. Information can be shared via the objects enqueued in the shared queue and also via the order in which the objects were enqueued. When a process dequeues an item from the queue, it gets access to the content provided by a potentially different process; therefore, the C-PRED of each successful dequeue operation is the enqueue event of the dequeued item. To maintain the order of the enqueue and dequeue operations, the C-PRED of an enqueue event is the event that *last* modified the queue (i.e. the latest enqueue or dequeue event). The queue creation

is the C-PRED of the first enqueue event.

5.9.3 Shared Locks

Shared locks are provided to allow for user process synchronization. Users can create a shared lock and execute lock, unlock, and trylock operations on it. These operations are also considered process communication events although they don't directly share any data. The C-PRED is maintained such that the chain of lock and unlock operations on a shared lock can be reconstructed. More specifically, a lock (or successful trylock) operation will be a predecessor of the unlock operation and the unlock operation will be a predecessor to the subsequent successful lock or trylock.

5.9.4 Deleted Objects

A problem arises with setting the C-PRED of events accessing shared state objects that have been deleted. Ideally, we would like to set the C-PRED of such an event to be the object's deletion event; however, this would require us to keep the deleted objects in the shared state forever. We do not store these objects and instead fill in the record's C-PRED field in the post-collection processing step by finding the deletion event for the appropriate object and setting the C-PRED field to that event's ID. Since multiple delete operations could have been issued for the same object concurrently, we are interested in the delete operation that actually succeeded, which will be captured by the event's status field. If the delete event is not found because the deletion event occurred a long time ago and has been archived, this field will remain blank.

5.10 File System

In the Aeolus distributed platform, sensitive user data can be stored persistently on designated storage nodes. Each storage node runs an Aeolus FS *manager*, which is responsible for servicing file system requests on data stored in the node's underlying file system. Each Aeolus file and directory is associated with an immutable secrecy

and integrity label, specified at creation time. Before data can be read from or written to a file or directory, Aeolus FS checks these labels and ensures that the requesting user process is authorized to perform these operations. In the current implementation, the label information is stored in a separate hidden metadata file (co-located with the data file).

Users can communicate with specific storage nodes by invoking corresponding file system methods of the Aeolus API and passing the hostname of the desired storage node as an argument to these calls. When a request is submitted for a given storage node for the first time at a compute node, a new Aeolus FS *client* is created to communicate with the Aeolus FS manager running on the specified node (there will be one client running on a given Aeolus node for each storage node that a user process on that node has communicated with). The Aeolus FS client submits file system requests to its Aeolus FS manager. To improve performance, the Aeolus FS client caches the information received from the manager. The client cache stores file content and labels based on the file pathname. If the data for a file is present in the cache upon a file read operation, the client checks the file labels and returns the data from the cache if the read is permitted. All file writes and directory updates (i.e. file and directory creations and deletions) are forwarded to the server. To ensure causal consistency, the server invalidates files in client caches by piggybacking a list of file names that have been modified in each client reply (the server also sends periodic invalidations to the clients). File stream operations are treated as block reads and writes; more specifically, when a file stream is open for reading, the entire file content is fetched to the client and writes to a file stream are accumulated locally and only sent to the server when the file stream is closed (as one file write).

We need to capture dependencies between events of different user threads that communicate by observing and updating data stored in the file system. More specifically, we want to set the C-PRED of each file read and write event to be the event that corresponds to the write operation that they observe (we treat directory content listing and file creations and deletions as reads and writes to the directory, respectively). Therefore, we would like to know the latest write event of each file or directory. One

possibility would be to keep this information in the metadata file that currently just stores the labels (this is similar to the shared state approach); however, this solution would require an additional write of the metadata file per each data file write operation (each file write would need to update the metadata file to reflect the latest write event). To avoid the cost of two disk writes, we have the Aeolus FS manager log its own events and correlate these events with those at the user side compute nodes. We describe this approach in detail below. Figure 5-4 shows the correlation of events recorded at the file system server and client side for file read and write operations described below.

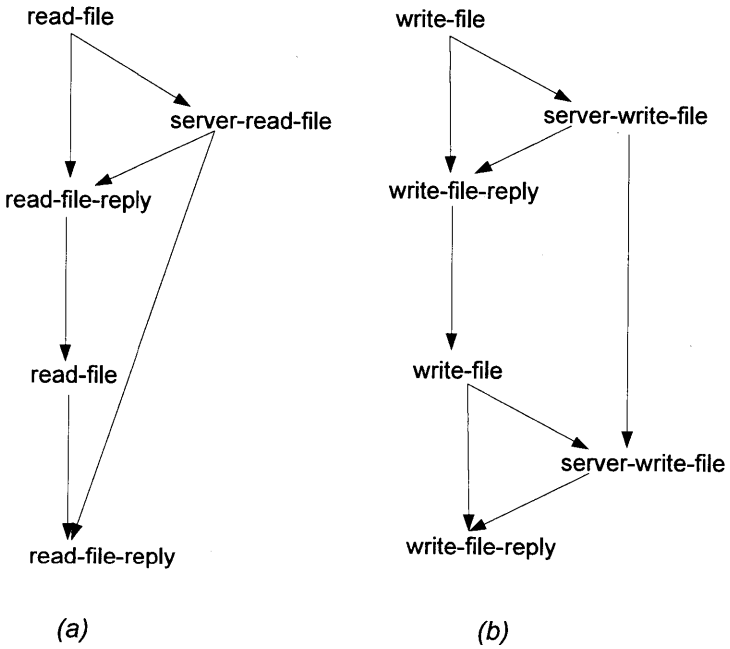

Figure 5-4: File System Events Correlation. (a) Events generated by two consecutive reads of the same file. The first read is cached at the client. (b) Events generated by two consecutive writes to the same file. The two write events at the server are correlated in the post-collection step.

5.10.1 Client Side

Each file system operation invoked through Aeolus corresponds to two events at the file system client side (i.e., in the invoking user thread). The first event represents the

file system operation request (e.g., file read request). This event includes information about the file involved (e.g. the file path name and file labels if the file is being created) but does not include application level data, such as the content written to or read from the file. The second event represents the file system reply and includes any Aeolus-relevant information returned from the file system to the client; for example, its status indicates whether the operation failed at the server, which could be due to a violation of information flow control rules. This event has two predecessors: its P-PRED is the request event and its C-PRED is the event at the file server that the reply observes.

5.10.2 Server Side

Aeolus FS manager receives concurrent file system requests from its clients. For each given file, it serializes the requests for this file such that there are no concurrent writes or concurrent reads and writes (i.e. only concurrent reads are allowed). Therefore, the Aeolus FS manager observes the exact order of file write operations for each given file and for each file read operation it knows what write operation preceded it (since reads and writes do not run concurrently).

The order that the manager observes is reflected in the order of its logged events. For each client request, the manager logs an event after the execution of this request. This event has as its predecessor the request event from the client. The manager then forwards the ID of its server event back to the client where that is set as the predecessor of the reply event. The client also stores this ID in its cache, so that any read from the cache has this event as its predecessor. Since operations are serialized at the server, we can reconstruct the predecessors of each server event in the post-collection step: for each file operation, we can find the latest update event at the server stored earlier in the log.

5.11 Use of I/O Devices

Users can write or read from various I/O devices. I/O devices are considered to be outside of our system. To ensure that sensitive data cannot be leaked outside the system, writing to an I/O device is only allowed if the security label of the user thread is null. To ensure that low-integrity data does not influence a high-integrity user thread, reading from an I/O device is only allowed if the integrity label of the user thread is null. Both operations correspond to one event record in the log, which denotes whether the user was reading or writing to a device outside the system. In the current prototype, we do not include information about the device being accessed; however, this can be included in the future.

5.11.1 Application Level Event Creation

Applications can include additional events in the audit trail by calling the `createEvent` method of the interface to our auditing sub-system. This method creates an event containing the user-supplied information and sets its P-PRED to the latest event in this user thread.

Chapter 6

Implementation

In this chapter we describe our audit trail collection and storage strategy, as well provide more details about what information is actually stored in an event record. More details on the implementation of the collection data structures can be found in Appendix B.

6.1 Collection and Storage

6.1.1 Collection

The audit trail collection framework consists of a *central logging service* responsible for aggregating event records generated across the entire system and *logging clients* responsible for sending local event records to the central service.

As illustrated in Figure 6-1, each platform instance runs a logging client that ships the event records generated by its user threads' activities to the central logging service via Java RMI. To optimize performance, event records are sent to the logging service *periodically* in batches, instead of sending an event record as soon as it is created. This strategy may lead to some events being lost if a node fails; we leave the repair of the audit trail in the presence of failures as something to be dealt with in future work (see discussion in Chapter 9).

Therefore, event records are stored locally by each platform instance's local *log*

collector. The log collector accumulates event records based on the ID of the user thread in which the corresponding user operation ran. When an event record is created, it is appended to the collection of event records generated by its user thread. This allows event records generated concurrently by different user threads to be stored in parallel.

When the records are shipped to the central logging service they are assembled into a structured collection that maps the user thread ID to a list of its event records. The event record shipment is scheduled at a fixed time rate, as opposed to shipping when a given record volume has been reached; the rate is a configuration parameter, e.g. 1 second. Along with the event records, each shipment also includes the ID of the sender platform instance and a system timestamp: all the events in this shipment will be stored with this timestamp to reduce performance overhead of querying the system clock every time an event occurs.

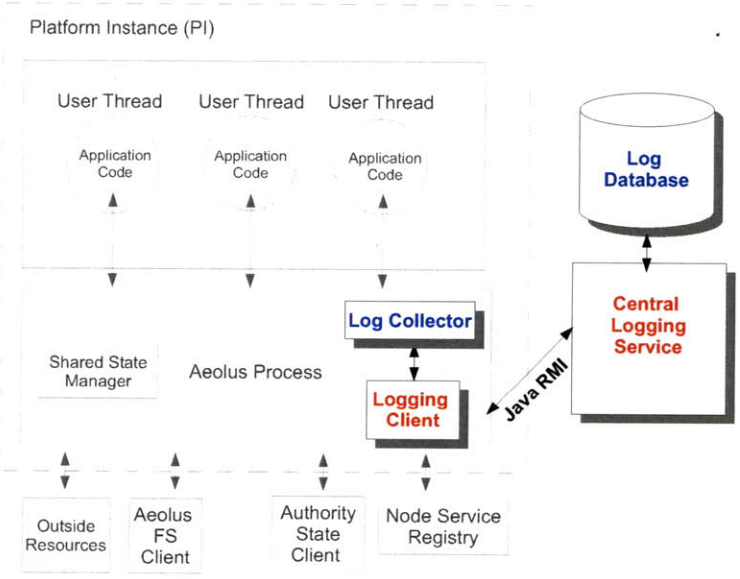

Figure 6-1: Audit Trail Collection and Storage Framework

6.1.2 Storage

After arriving at the central logging service, the event records are processed and stored in a logically centralized database. The processing step includes filling in the timestamp field, as well as the P-PRED if it's missing (the next section explains why this predecessor is not stored whenever possible). However, this field is easily derivable since the event records are stored and shipped in the order that they were created by the user thread; therefore, with some exceptions, the P-PRED is the previous event in the event record list associated with the given user thread ID.

Storing logs in the database provides more efficiency and flexibility for data processing. We store the records in a MySQL relational database and the Neo4j graph database; the later is good for dependency graph generation [1]. Currently, a database entry includes all the information stored in an event record, as well as the ID of its platform instance and user thread.

Archiving and protecting the data in the database is left for future work, as discussed in Chapter 9.

6.2 Event Records

This section describes our strategy for creating globally unique event IDs and what event information is excluded from an event record to reduce the logging overhead.

6.2.1 Event IDs

We use a uniform scheme for creating structured unique IDs for the events in our system. The structure of event IDs provides information about the context of each event within our system (e.g. the node or platform instance at which the event ran), which can be used in recovering from partial audit trail data loss (e.g. due to a node failure) and provides additional user-readibility of audit trail records.

Most recorded events are generated at the invocation of an Aeolus API method in a user thread of a platform instance running on one of our system nodes. Each platform

instance has a system-wide unique ID (no platform instance will ever run with the same ID in the system). Within each platform instance, each user computation runs inside a user thread whose ID is unique *relative* to the platform instance; however, user threads can be reused and an event from the same user thread may or may not be part of the same user computation. To identify events within a user thread, we keep a counter that is incremented after each recorded event. To avoid endlessly increasing this event counter, we reset it every time a user thread is reused and associate instead an *incarnation* number with each user thread to reflect how many times it has been reused.

Therefore, most events can be uniquely identified by the following 4-tuple:

`(PI ID, user thread ID, user thread incarnation #, event counter).`

However, there are several events in the system that are not recorded inside a user thread. These are special events recorded at the creation and deletion of a system node, the launching and shutdown of a platform instance, and the Aeolus FS manager events. Since each platform instance has a unique ID, we have two special event IDs for the PI launch and shutdown events, namely:

PI Launch: `(PI ID, -, -, 0)`
PI Shutdown: `(PI ID, -, -, 1).`

Furthermore, if we associate the node hostname with the unique ID of its start-up PI, we obtain the following unique IDs for a node creation and node deletion events:

Node Creation: `(hostname, start-up PI ID, -, 0)`
Node Deletion: `(hostname, start-up PI ID, -, 1).`

The file system manager events need to be unique across storage nodes and storage node relaunches. We differentiate events at different storage nodes by including the node hostname in the ID. We use an incarnation number for each relaunch of the file system manager (this is a counter persistently stored in a special file system configuration file on each node). Finally, the file system manager maintains an event

counter for its events and so the IDs of events logged by the file manager look as follows:

Aeolus FS manager events: `(hostname, incarnation #, -, event counter)`

6.2.2 Eliding

We avoid storing information in event records whenever possible in order to reduce event processing costs and the size of the event records. Since event records are sent to a central log collector, the size of the record affects the communication overhead.

Primarily, we do not include information that can be deduced from the audit trail history. For example, we can derive the labels of each user thread event since they are stored when the thread first starts running and any further label modifications are recorded in the audit trail. Therefore, it is not necessary to record the return value of an authority operation, such as a declassify or endorse request. On the other hand, we do log the return values of all non-deterministic operations, such as the IDs of newly created principals, tags, closures, and shared state objects, which are generated randomly. We also record whether an operation succeeded or failed, since unpredictable non-Aeolus failures are possible.

Due to the way that the event records are collected (as discussed in Section 6.1.1), we often do not need to store the P-PRED of these events. This information can be re-derived at the central log collector when the event records are processed after the shipment.

Finally, we do not record the timestamp in the event records. This reduces the size of the events and avoids the expensive read of the system clock for each event. Instead, the timestamp is computed when the PI's logging client sends events to the central logging service, as described in the Section 6.1.1.

Chapter 7

Auditing Case Studies

In this chapter, we present audit trail records collected during the run of several sample applications. Our first example is a small application built solely for demonstrative purposes. It shows uses of the shared state, forks, and remote procedure calls. Our next example is a simple medical clinic application. It demonstrates uses of authority and persistent data storage via the file system. We present the collected audit trails as dependency graphs. These graphs were generated using the open source tool *graphviz* [7] from the data stored in the log database.

7.1 Demo Application

Our first example is a small distributed client-server application built for demonstrative purposes only. The application runs on two nodes: the service runs on `26-2-69.dynamic.csail.mit.edu` and the client runs on `eigenharp.csail.mit.edu`.

Table 7.1 shows the pseudocode of the program executed at the service and the client and Figure 7-1 shows the graph of event records stored during the run of this application. The service program registers the *RpcTest* service with Aeolus, writes to an I/O device, and waits for clients to connect. The client program runs in two threads. The starter thread first creates a shared queue, forks a second thread providing it the handle to the queue, and then waits to dequeue a value from the shared queue. The forked thread3 places an object in the queue and terminates.

Table 7.1: Demo Client-Service Program

Server – Side
1. RpcTestService service = new RpcTestService() 2. `registerService`("RpcTest", service) 3. `writeToIODevice`(screen, "Started service...")
Client – Side
1. handle = `createSharedQueue`(sLabel=empty, iLabel=empty) 2. AeolusCallable c = new TestCallable(handle) 3. `fork`(c, p2) 4. value = `waitAndDequeue`(handle) 5. RpcTest serviceStub = (RpcTest) `getService`(26-2-69.dynamic.csail.mit.edu, "edu.mit.csail.aeolus.testing.RpcTest") 6. result = serviceStub.`testMethod`(value) 7. `createEvent`("APP-EVENT-RECEIVE-RPC-REPLY", ret-val = result)
TestCallable(handle)
1. value = new Integer(5) 2. `enqueue`(handle, value)
RpcTestService.testMethod(value)
1. `writeToIODevice`(screen, value) 2. return 1

After it receives the value from the queue, the starter thread obtains a local stub for the service *RpcTest* on host `26-2-69.dynamic.csail.mit.edu` and invokes the remote method *testMethod*. The method runs at the server in a new user thread; it writes to an I/O device and returns to the client. Finally, since Aeolus does not log application values such as parameters and results of remote procedure calls, the client creates an application event to store the value returned by the remote method invocation.

The IDs of each event in the graph provide information about what platform instance and user thread a given event occurred in. For example, the CREATE-SHARED-QUEUE event was created in platform instance with ID 7 and user thread with ID 13. Each node also includes the parameters and the return value stored with each event record. For example, the PI-LAUNCH event parameters specify the ID of the created platform instance and its root principal ID; the parameter to FORK speci-

fies the principal to run the forked thread with (in this case the principal ID is 3); the return value of the CREATE-SHARED-QUEUE event stores the handle of the new shared queue and this handle becomes the parameter of the ENQUEUE and WAIT-AND-DEQUEUE events (here the handle value is 13); the parameters of the GET-SERVICE and SEND-RPC events store information about what host the service runs on, the service name, and the method name for the latter (in this case the hostname is `26-2-69.dynamic.csail.mit.edu`, the service is *RpcTest*, and the method name is *testMethod*).

The edges of the graph show the dependencies among the recorded events. We can see that the WAIT-AND-DEQUEUE event depends both on the previous event in the same thread (namely, FORK) and on the ENQUEUE event of the forked thread. In the case of the remote procedure call, we can see that the RECEIVE-RPC event at the server depends on the SEND-RPC at the client and the RECEIVE-RPC-REPLY event at the client has the SEND-RPC-REPLY event as its predecessor. The RECEIVE-RPC also has the REGISTER-SERVICE event of the requested service as its predecessor.

7.2 The Medical Clinic

This section presents the data collected from a simple medical clinic application. Our clinic has three types of members: the patients seeking treatment at the clinic, the administrators responsible for registering patients and assigning their doctors, and the doctors responsible for examining the patients and updating their medical history. Tables 7.2 and 7.3 show the pseudocode for the patient registration and the doctor's patient examination subprograms.

The following steps occur when a new patient, Alice, is registered at the clinic using the registration program (i.e. via a call to `registerPatient`(*Alice*) of Figure 7.2). First, a new principal ID, P_{Alice}, is generated to represent Alice along with a tag, $T_{AliceMedical}$, to protect her medical history. A new file is created to store Alice's medical history data; this file's security and integrity labels include the tag $T_{AliceMedical}$. A special principal is then created to represent Alice's current physical,

Table 7.2: Patient Registration by a Clinic Administrator

`registerPatient`($Patient$)
// registration runs with P_{Admin}
1. $P_{Patient}$ = `createPrincipal`()
2. $T_{PatientMedical}$ = `createTag`()
3. `createFile`("/patients/$Patient$Record", sLabel = $T_{PatientMedical}$, iLabel = $T_{PatientMedical}$)
4. $P_{PatientDoctor}$ = `createPrincipal`()
5. `delegate`($T_{PatientMedical}$, P_{Admin}, $P_{PatientDoctor}$)
// assign a doctor to patient
6. `actFor`($P_{PatientDoctor}$, P_{Doctor})
// update the all-patients file to include this patients's registration info
7. patientInfo = "$Patient$: $P_{Patient}$, $T_{PatientMedical}$, $P_{PatientDoctor}$"
8. `endorse`(T_{Admin})
9. `appendFile`("/patients/AllPatients", patientInfo)
10. `removeIntegrity`(T_{Admin})

$P_{AliceDoctor}$, and $T_{AliceMedical}$ is delegated to this principal to allow it to review and update her medical history. Finally, an available physician, Bob, is assigned to treat Alice; this is achieved by letting Bob's principal, $P_{DoctorBob}$, act for Alice's physician (each doctor has his or her own principal ID in the clinic). Note that if Alice switches to a different doctor, the act-for link between Alice's physician and Bob is revoked.

When registering a patient, the administrator also updates the all-patients file to store Alice's administrative information, such as the principals P_{Alice} and $P_{AliceDoctor}$. Each administrator has its own principal, P_{Admin}, and all administrators share the tag, T_{Admin}, which is stored in the integrity label of the all-patients file, such that only administrators can update this file (however, doctors can read it).

During Alice's medical exam, her doctor reviews the all-patients file and switches to $P_{AliceDoctor}$ via a local call operation. $P_{AliceDoctor}$ is authoritative for the tag $T_{AliceMedical}$, which the doctor adds to his integrity label in order to update Alice's medical history. This is illustrated in Figure 7.3.

Figure 7-2 and 7-3 show the audit trails collected from each subprogram during the registration of one patient, Alice, and her medical examination by the assigned physician. We only depict the events generated by the programs in Tables 7.2 and

Table 7.3: Patient Examination by a Doctor

`examinePatient`(*Patient*)

// examination runs with P_{Doctor}
// doctor first reads the patient registration data for *patName*
1. ($T_{PatientMedical}$, $P_{PatientDoctor}$) = `readFile`("/patients/AllPatients")
2. AeolusCallable c = new AeolusCallable() {
3. `addSecrecy`($T_{PatientMedical}$)
4. patientHistory = `readFile`("/patients/*Patient*Record")
5. newHistory = "The examination revealed... Recommended treatment...."
6. `endorse`($T_{PatientMedical}$)
7. `appendFile`("/patients/*Patient*Record", newHistory)
8. `removeIntegrity`($T_{PatientMedical}$)
9. `declassify`($T_{PatientMedical}$) }
10. `call`(c, $P_{PatientDoctor}$")

7.3 and do not present the node creation, platform instance, and user thread launch events; we condense all missing events from the user thread into one event, the LAST-USER-THREAD-EVENT; the latest authority update at the authority server is denoted as LATEST-AUTH-UPDATE. Each program runs in single user thread. No other code executes in the system while these programs are running.

Figure 7-2 shows the results of registering Alice at the clinic. First Alice's principal and medical tag are created. Each of these operations is an authority update and corresponds to three events: two events at the client (the request and reply) and one event at the authority server. The principal and tag IDs are captured by the return values of the CREATE-PRINCIPAL-REPLY and CREATE-TAG-REPLY and have, respectively, the values of 3 and 15. The file path of Alice's medical data is captured by the parameter to the create file operation. As described in Chapter 5, this operation corresponds to two events at the file client node: the CREATE-FILE request event and the CREATE-FILE-REPLY event and one event at the file server, FS-CREATE-FILE. The labels of the created file are also stored in the event parameters; they include Alice's medical tag, 15. Next, we see the creation of Alice's doctor principal; its ID is 4, as indicated by the return value of the reply event. Alice's medical tag, with ID 15, is then delegated to Alice's doctor principal. The assignment of a doctor to Alice is

captured by the ACT-FOR event, which creates an act-for link between the doctor with principal ID 2 and Alice's doctor principal with ID 4. Finally, the administrator adds the administrator tag (with ID 10) to his integrity label to update the all-patients file.

After Alice is registered, she is seen by her doctor who runs the patient examination program. The audit trail events are depicted in Figure 7-3. The doctor first reads the all-patient file and then makes a call operation to switch to Alice's doctor principal (with ID 4 depicted in the parameters to CALL). During the execution of the call, the doctor reads and updates Alice's medical file. He adds Alice's medical tag with ID 15 do his secrecy and integrity label to perform these operations (as depicted by the ADD-SECRECY and ENDORSE events). The ENDORSE event, as well as the CALL, REMOVE-INTEGRITY, and DECLASSIFY events, is an authority lookup and has the latest authority update event of the authority client as its C-PRED. Finally the return from the call is represented by the CALL-RETURN event.

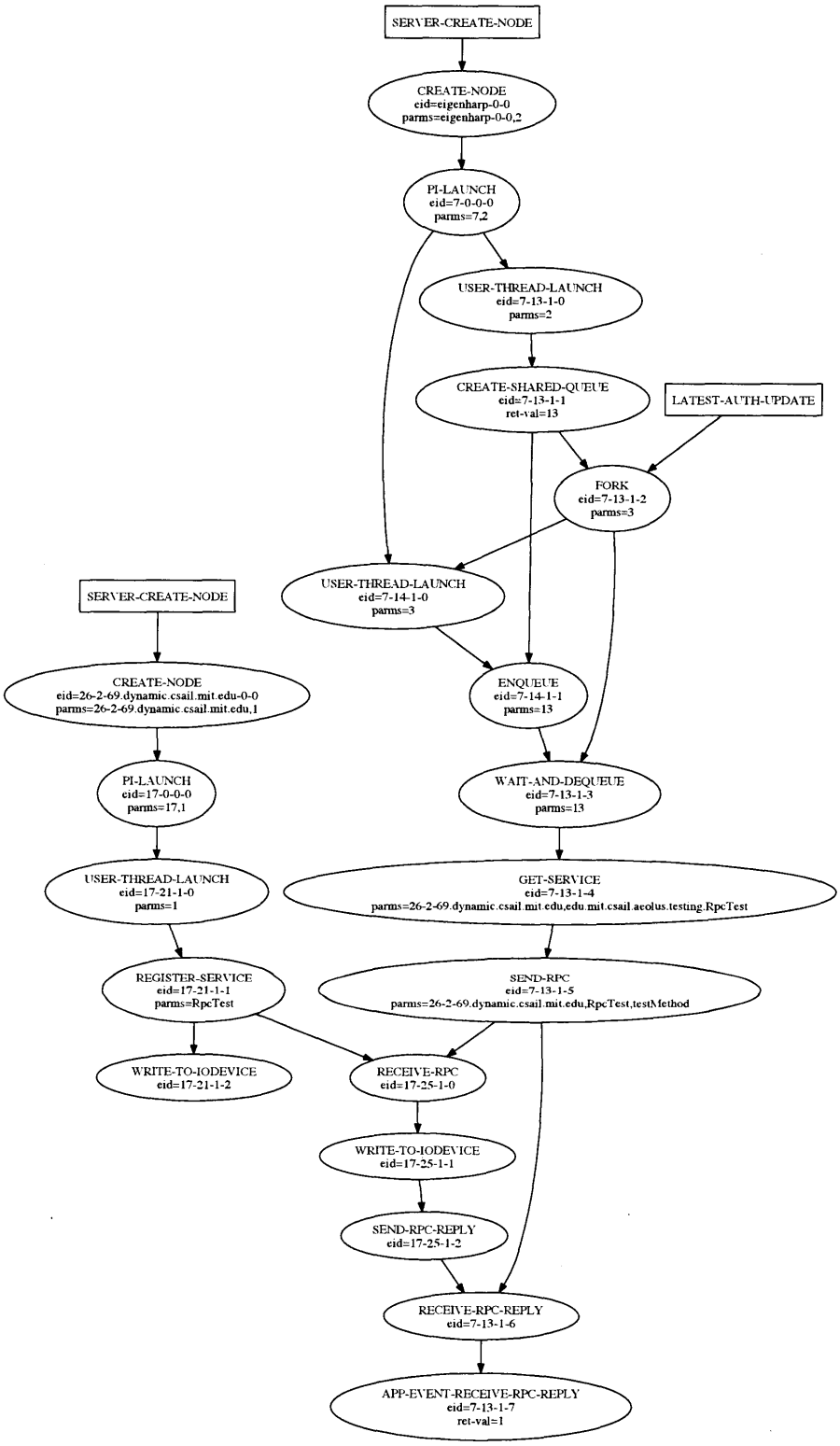

Figure 7-1: Audit Trail Generated By the Client-Service Demo Application

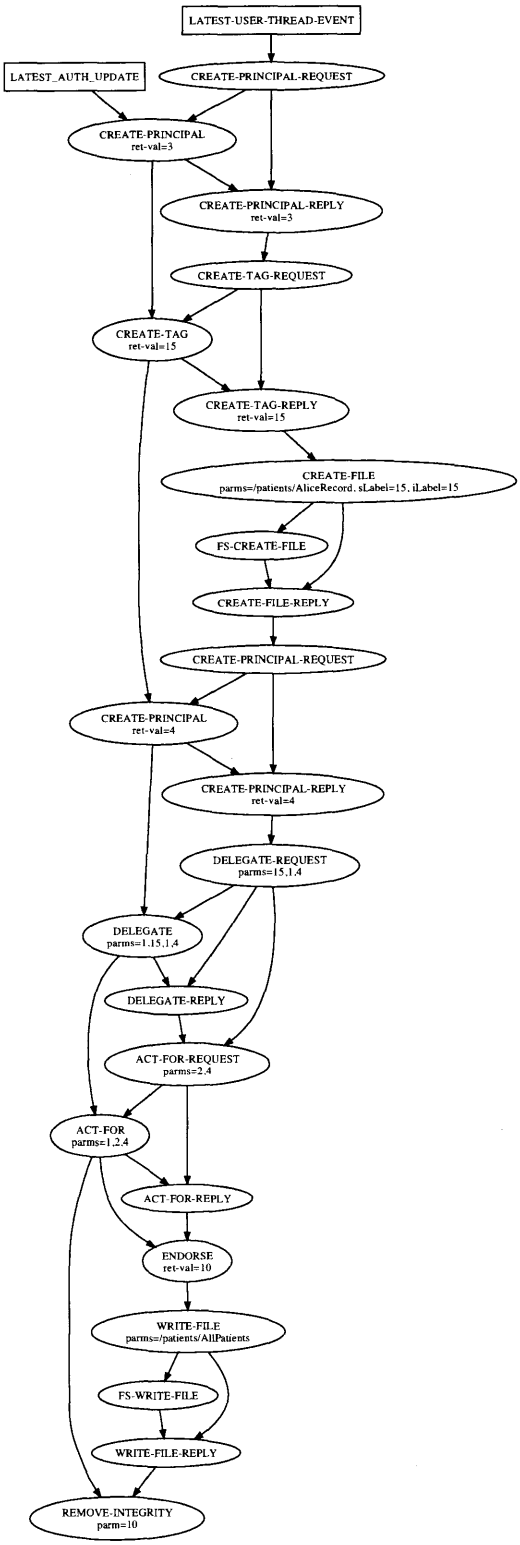

Figure 7-2: Audit Trail Generated By the Patient Registration Program

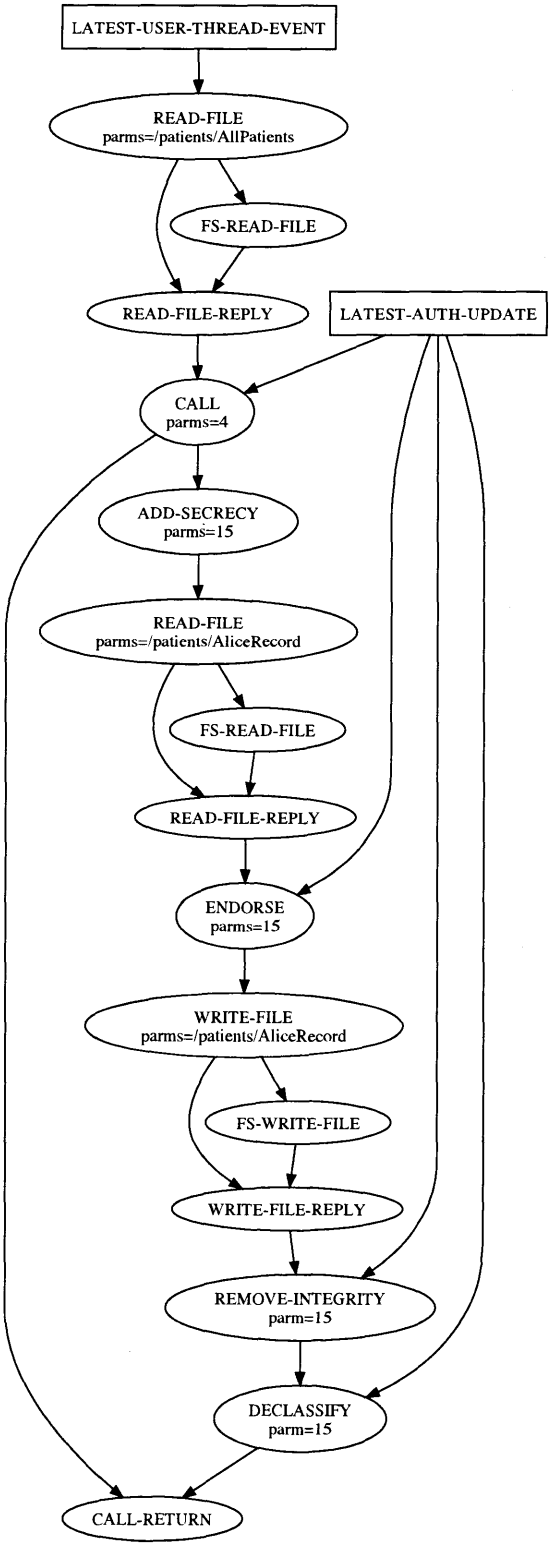

Figure 7-3: Audit Trail Generated By the Doctor Examination Program

Chapter 8

Performance Evaluation

Aeolus audit trails monitor all user activity in the system and provide means to reconstruct the chain of events leading to a given system state, by recording every user call to the Aeolus API (as well as occasional system events) and maintaining links between the recorded events. This poses additional overhead in serving Aeolus API calls. In this chapter we examine the performance characteristics of collecting and storing the audit trail content. We present a series of micro-benchmarks to evaluate the logging overhead on each individual operation type. We also include results from several macro-benchmarks that examine the overall impact of logging on system performance.

8.1 Experimental Setup

All experiments described in this chapter were performed on identical machines running Ubuntu 9.10 with OpenJDK Java version 1.6.0_0 and Eclipse 3.5.1 installed. Each system has 2.83 Ghz Intel Q9550 quad core processors and 4GB of RAM. The machines were connected via a 100Mbps switch with the network roundtrip time of less than 1 millisecond. In order to measure steady-state performance, Java recommended micro-benchmarking practices [16, 4] were followed. In particular, all experiments included a warmup phase intended to trigger all initializations and compilations. We ensured that no code path was taken for the first time during the timing

phase to avoid deoptimization and recompilation effects. In order to avoid dead-code elimination, we also ensured that the results computed by the benchmark were somehow used (otherwise the compiler can eliminate the computation if it determines that the computation cannot affect the output). In order to avoid full garbage collection to take place during the timing phase, we clean up the JVM before each benchmark (using *System.gc*()) and ensure that sufficient memory is available (the JVM `-Xmx` parameter was set to 2GB during the runs). The `-XX:+PrintCompilation` and `-verbose:gc` parameters were set during the runs to ensure that the compiler and the garbage collector are not performing unexpected tasks during the timing phase. The `-XX:CICompilerCount=1` argument was also used to prevent the compiler from running in parallel with itself. In order to avoid CPU caching effects, tests were run with varying sizes of the data set, whenever appropriate. Each benchmark was run multiple times, always in a separate JVM. The number of iterations was set such that the total time exceeds a couple of seconds whenever possible (most average values were computed over 100,000 runs). Finally, noise was reduced as much as possible during the measurements.

8.2 General Costs

Since all user interactions with Aeolus are monitored, there is a generic cost of creating and storing an event record associated with every call to the Aeolus API. Although event records vary in size (e.g., due to the number and size of the user-supplied parameters), there is a common cost associated with storing the record in local memory, shipping it to the server, and finally storing it in the database. On the other hand, the cost of capturing and storing event predecessors varies significantly among different types of Aeolus API calls. For example, we maintain additional information for each shared state object in order to track its update history and we create special log entries at the file server so that we can capture the order of file system events. The micro-benchmarks presented next describe the costs associated with logging each individual operation type. In order to maximize the logging overhead, we try to mini-

mize the execution time of each operation; for example, we supply small-sized objects to shared state operations and use null labels whenever appropriate.

8.3 Local Forks and Calls

8.3.1 Forks

Aeolus provides the `Fork` method, which allows programmers to fork a new user thread running some provided code with a given principal. An event record is created by the `Fork` method before the new user thread starts executing and an additional event record is registered at the new user thread launch. In addition, in order to correlate the events in the parent and child threads, the ID of the fork event is carried over to the child process where it becomes the predecessor of its first event (namely, the user thread launch event). Table 8.1 shows the average request time for executing `Fork` requests with and without logging. This time includes the allocation of an unused user thread and the execution of the code inside that thread. The code simply outputs a character to the screen. Since the processing of a fork request itself is on the order of several microseconds, the experiment measures the total time of executing 1000 consecutive fork requests and then computes their average. Therefore, a small overhead is added due to the synchronization of the parent and child thread; namely, the parent thread needs to be notified that the child has finished in order to fork the consecutive thread (this is achieved through the use of a static variable).

Since any user process can switch to the public principal, P_{public}, running fork with this principal is less expensive - the switch is authorized without consulting the authority client. Similarly, any principal can fork with the its own pid. When requesting to run with a different principal, a request to check the authority of the caller process is sent to the authority client, whose reply carries both the return value of the authority lookup, as well as the ID of its preceding authority update event, which creates some additional overhead in performing the authority check.

We found that the logging overhead of the `Fork` operation is statistically insignif-

icant.

Table 8.1: Average Request Time (ms) of Fork

Operation	Aeolus	Aeolus w/ Logging
Fork to pid	0.028	0.03
Fork to P_{public}	0.023	0.024

8.3.2 Calls

Programmers can also run code with different authority in the same user process using the `Call` method. Two event records are created for this operation - before and after the code invocation. Similarly to the `Fork` call, switching to a non-public principal requires an additional authority lookup that carries logging information from the authority client. In the case of the `Call` operation, however, the event ID of the predecessor of the first event in the invoked code can be easily accessed (since the method is invoked in the same user domain). Table 8.2 presents the average of performing a `Call` operation with and without logging. The executed code simply prints a character to the screen. Averages were computed over 10,000 runs. Similarly to the `Fork` operation the switch to a public principal is always authorized and requires less processing time. As with the `Fork` operation, the logging overhead is very small (less than 10 microseconds per call).

Table 8.2: Average Request Time (ms) of a Call

Operation	Aeolus	Aeolus w/ Logging
Call to pid	0.02	0.028
Call to P_{public}	0.017	0.028

8.4 Remote Procedure Calls

Users can invoke methods on remote services registered with Aeolus by retrieving a service proxy from Aeolus at the client side - this proxy serves as a local reference

to the remote service object. A method invocation on this proxy triggers Aeolus to send an RPC request from the client node to the node hosting the service. In order to correlate the events at the client and server side, both the RPC request and reply carry predecessor event IDs. Two events are recorded at the client side: when sending the RPC request and upon the receipt of the reply. Respectively, two events are also recorded at the server side: upon the receipt of the RPC request and when sending the RPC reply. In addition, some logging state is maintained at each node service registry to correlate the registering and access of user services.

For this benchmark, we created a service that simply maintains an integer counter. A client can invoke `increment(value)` to increase the counter by the specified amount. Our experiment consisted of a client and a server running on two separate machines. Table 8.3 shows the average round-trip latency of the `increment` method call with and without logging. Logging adds a small overhead of roughly 80 microseconds per one RPC. The presented values were averaged over 1000 invocations (each call simply incremented the counter by 1). We expect the costs of an Aeolus RPC operation to be higher in our final implementation; however, this cost increase is not related to the logging costs and would only make the logging overhead comparatively smaller.

Table 8.3: Average Round-Trip Latency (ms) of a Basic RPC

Aeolus	Aeolus w/ Logging
2.37	2.455

8.5 Shared State

The shared state provides a way for user processes of the same platform instance to exchange information and synchronize. Every shared state operation corresponds to one event record. There are three ways of communicating via the shared state: through the use of shared objects, shared queues, and shared locks. In order to correlate access and modifier events of these objects, additional state is maintained at each object to capture its latest modification event. Therefore, calls that modify

the shared state objects also carry the IDs of the modifier events and each reply from the shared state manager carries the ID of the predecessor event.

8.5.1 Shared Objects

Every update of the shared object content needs to modify the associated field storing the latest modifier event ID. Every read of the file content also reads this field. Since access to shared objects is synchronized through the use of read-write locks, no additional synchronization is necessary to maintain this field.

Table 8.4 shows the cost of basic operations on a shared object that contains an integer and no labels with and without logging. These values were averaged over 100,000 iterations. In all cases, the logging overhead is very small (on the order of a microsecond).

Table 8.4: Average Request Time (ms) of Basic Operations on Shared Object

Shared Object Operation	Aeolus	Aeolus w/ Logging
CreateObject	0.0134	0.0141
GetObject	0.0116	0.0124
ReplaceObject	0.0113	0.0118

8.5.2 Shared Queues

The enqueue and dequeue operations are also synchronized, which removes the need of synchronizing the update of the last queue modifier event. However, each queue entry needs to store the ID of its enqueue event.

Table 8.5 shows the cost of basic operations on a shared queue with no labels that contains integer-size objects with and without logging. These values were averaged over 100,000 iterations. In all cases, the logging overhead is very small (on the order of a microsecond).

Table 8.5: Average Request Time (ms) of Basic Operations on Shared Queue

Shared Queue Operation	Aeolus	Aeolus w/ Logging
Enqueue	0.0124	0.0135
Dequeue	0.00019	0.00024

8.5.3 Shared Locks

Each lock also has an associated last-modified field which is updated when a lock is acquired or released. Since lock acquire and release operations serialize lock state modifications, there is no need for additional synchronization.

Table 8.5 shows the cost of basic operations on a shared queue with no labels that contains integer-size objects with and without logging. These values were averaged over 100,000 iterations. In all cases, the logging overhead is very small (less than a microsecond).

Table 8.6: Average Request Time (ms) of Basic Operations on Shared Lock

Shared Lock Operation	Aeolus	Aeolus w/ Logging
Lock	0.00047	0.00085
TryLock	0.00046	0.00084
Unlock	0.00046	0.00083

8.6 Authority State

Users interact with the authority state via authority update and lookup operations.

8.6.1 Authority Lookups

Authority lookups are processed at the local authority client and do not pass any additional information to the client since they are not reflected in the authority state modifier chain. As part of its reply, the authority client includes the ID of the latest authority update event that becomes the predecessor of the given lookup event. Table 8.7 shows the average time of a successful `Declassify(t)` operation with and without

logging. A call to `Declassify` includes an authority lookup operation, which checks if the user process is authorized to remove the given tag t from its secrecy label. This `Declassify` succeeds and required information is in the authority client cache during the measurement. The benchmark first creates the tag so that the predecessor ID passed back when logging is turned on is not null (this ID will be the ID of the tag creation event). The average was computed over 100,000 runs. The logging overhead is very small (less than a microsecond).

Table 8.7: Average Request Time (ms) of an Authority Lookup Operation

Aeolus	Aeolus w/ Logging
0.00057	0.00076

8.6.2 Authority Updates

An authority update corresponds to two recorded events at the client (the request and the reply). The ID of the authority update request event needs to be carried to the authority server. The authority server logs an additional event. The ID of this event is stored in the database together with other information associated with this update event. This ID is also included as part of the server's reply back to the client. Table 8.8 includes the average value of a `createPrincipal` operation computed over 1000 iterations. The network roundtrip time was found to be under 1 millisecond.

Table 8.8: Average Request Time (ms) of an Authority Update Operation

Operation	Aeolus	Aeolus w/ Logging
createPrincipal (total)	1.692	1.714

8.7 File System

The Aeolus FS manager runs on Aeolus storage nodes and manages the data stored in the underlying file system. Namely, it checks that all the incoming file system requests

are authorized by verifying the file secrecy and integrity labels. The manager serializes all the operations on the same file, such that there are no concurrent writes to the file or concurrent reads and writes (only file reads can run concurrently). For each file operation it executes, the manager creates an audit trail event record that captures the information about the executed operation. It sends the ID of this event back to the client.

At the file system client node, each file system operation corresponds to two event records: one representing the file operation request and one the reply. The ID of the request event is forwarded to the server, where it becomes the predecessor of the sever logged event. The server event ID is sent back to the client, where it becomes the predecessor of the reply event. The client caches the file read operations. This cache includes the content of the file, as well as the ID of its read event at the file server. If the information is found in the cache, this ID is set as the predecessor of the file read reply event.

The number of created events and the way they are correlated is the same for all file operations except for cached file reads. Our experiment measures the logging overhead on file writes and cached file reads. The client and the server are located on two separate machines. Table A.18 shows the average value of a file read operation on a 5KB file when the file content is found in the cache and the average value of a file write operation (5KB of data are written to the file in each call). The averages were computed over 1000 iterations. The cached file read operation is very fast and the logging overhead is very small (under 1 microsecond). The logging overhead on the file write operation is also small (around 15 microseconds).

Table 8.9: Average Request Time (ms) of Basic File Operations

File Operation	Aeolus	Aeolus w/ Logging
Read (cached)	0.00022	0.00084
Write (server)	0.4801	0.4957

8.8 End-to-End Evaluations

8.8.1 Medical Clinic

We implemented a simple medical clinic application to examine the overall logging overhead on a common application pattern involving authority updates and frequent access to the file system. A detailed description of this application can be found in Chapter 7. In this experiment we have an administrator principal that registers several patients and assigns them to a doctor and one doctor that later examines and updates all the patient files. In total, 1000 patients are first registered and assigned to a doctor and all their medical records are then examined and modified by their assigned doctor (for simplicity the clinic only has one doctor). The patient medical records are small files of 10KB. We only measure the time it takes one doctor to view and update all patient files. Each file update overwrites the entire file so that the patient files always remain the same size. Table 8.10 includes the average time it takes the application to run with and without logging on 1000 patient files. We found that logging created roughly 9% overhead.

Table 8.10: Average Runtime (s) of the Medical Clinic service

Aeolus	Aeolus w/ Logging	Overhead Factor
1.010	1.100	1.08

Chapter 9

Conclusions

This thesis has presented the design and implementation of audit trail collection for Aeolus, a distributed security platform based on information flow control. Aeolus audit trails are event logs that monitor the interaction of user application code with the Aeolus platform. In addition they can include application level events recorded using the provided user interface to the auditing framework.

Event records are collected on each system node and shipped to a centralized location, where they are stored in a database and processed. We correlate events by capturing any dependencies between them in the audit trail records. Each audit trail record stores links to its immediate predecessor events; this is achieved by uniquely identifying each event in the system. Audit trail records form event dependency graphs, where the nodes are the events and the edges represent the causality relationship between them captured by the predecessor links. From this information, we can easily reconstruct event chains by following the event predecessor links. For example, we can find all the events that could have led to a given system state.

A new prototype of the Aeolus platform has been developed in Java and includes the auditing collection framework. Our results show that logging imposes a small overhead on system performance.

This work has focused on the content, format, and the collection and storage mechanism of the audit trails. Subsequent work will mostly focus on the analysis and management of the collected data. We now discuss future project directions.

- *Protection.* We want to ensure that only authorized parties can get access to the data stored in the audit trail database and that no erroneous data can be added to the database. This can be achieved using information flow control to protect the database.

- *Archiving.* Events that occurred long in the past need to be moved from the working database to an archive. We need a criteria for determining when events can be removed; the archive files also need to be protected.

- *Reliability.* We need to replicate the central audit trail storage node to ensure that audit trail information will not be lost if this node fails.

- *Recovery.* Since audit trails are shipped to the central storage only periodically, some event records can be lost due to a node failure. We need to develop a strategy for recovering from this loss. First, we need a way of determining when missing events are permanently lost. If a loss is detected, we need to reconstruct the chains of events that have been broken. Event IDs will be useful for recovering some missing information. For example, they include enough information to discover the latest event of a user thread that has not been lost; they can also identify how many events are missing from some given user thread execution based on the value of the event counter.

- *Graphical User Interface.* We need to find ways to present the event dependency graphs to the users. These graphs can be very large; therefore, the interface needs to provide some way of focusing on some chosen events, while representing others in a condensed way. The interface should also handle node rearrangement and deletion.

- *Filtering.* We need to develop a graph algorithm for filtering out event records based on some criteria. If an event record needs to be removed (e.g. because it does not match the criteria of interest), its predecessor chains need to be changed appropriately (i.e. if event E is being removed and it's a predecessor of an event A, then $Pred(A)$ needs to be set to $Pred(A) - E + Pred(E)$). Since the

event dependency graphs can be very large, the algorithm could be parallelized to explore nodes concurrently.

- *Querying.* We need to understand what kind of queries are useful for analyzing the audit trail records and provide efficient algorithms for running them and representing their results. Additional data might be stored in the database event record entries to improve performance. For example, if a common query is to find all the events created by a given principal, we might want to store the principal ID associated with each event in the event database entry (currently the principal ID is only stored in the user thread launch event of a fork, a local call, and an authority closure call).

- *Detection.* We might want to provide tools to help detect anomalies in the audit trails. There are various attack signature-detection tools that look for a specific sequence of events that signals a security violation. We can provide a similar tool that would match a given chain of events to the events stored in the audit trails. We need to understand what patterns are relevant to our system.

- *Backtracking.* Audit trails maintain a history of system event execution and if a security violation is detected, we could use this information to backtrack to a valid system state. For example, we could store periodic snapshots of the authority state and replay authority updates from a given valid snapshot.

Appendix A

Audit Trail Events

This appendix describes what events are recorded for each Aeolus operation and what information they store. We do not include context-dependent information, such as the timestamp and success/failure status. Only normal case behavior is explained. The P-PRED is assumed, unless explicitly included.

Table A.1: Principals

Aeolus API Method	Event Records
`pid createPrincipal()`	e1. CREATE-PRINCIPAL-REQUEST e2. CREATE-PRINCIPAL (at server): *preds*={e1,*last*-auth-server-update}, *parms*={caller-pid}, *ret-val*={pid} e3. CREATE-PRINCIPAL-REPLY: *preds*={ e1, e2}, *ret-val*={pid}
`pid createPrincipalInNewCore()`	e1. CREATE-PRINCIPAL-NEWCORE-REQUEST: e2. CREATE-PRINCIPAL-NEWCORE (at server): *preds*={e1,*last*-auth-server-update}, *parms*={caller-pid}, *ret-val*={ pid } e3. CREATE-PRINCIPAL-NEWCORE-REPLY: *preds*={e1, e2}, *ret-val*={pid}
`pid2 createPrincipalInCore(pid1)`	e1. CREATE-PRINCIPAL-INCORE-REQUEST: *parms*={pid1} e2. CREATE-PRINCIPAL-INCORE (at server): *preds*={e1,*last*-auth-server-update}, *parms*={caller-pid, pid1}, *ret-val*={pid2} e3. CREATE-PRINCIPAL-INCORE-REPLY: *preds*={e1, e2}, *ret-val*={pid2}

Table A.2: Tags

Aeolus API Method	Event Records
`tid createTag()`	e1. CREATE-TAG-REQUEST: e2. CREATE-TAG (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid}, *ret-val*={tid} e3. CREATE-TAG-REPLY: *preds*={e1, e2}, *ret-val*={tid}
`tid2 createSubTag(tid1)`	e1. CREATE-SUBTAG-REQUEST: *parms*={tid1} e2. CREATE-SUBTAG (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid, tid1}, *ret-val*={ tid2 } e3. CREATE-SUBTAG-REPLY: *preds*={e1, e2}, *ret-val*={tid2}

Table A.3: Delegations and Revocations

Aeolus API Method	Event Records
`actFor(p1, p2)`	e1. ACT-FOR-REQUEST: *parms*={ p1, p2 } e2. ACT-FOR (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid, p1, p2} e3. ACT-FOR-REPLY: *preds*={ e1, e2}
`revokeActFor(p1, p2)`	e1. REVOKE-ACT-FOR-REQUEST: *parms*={p1, p2} e2. REVOKE-ACT-FOR (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid, p1, p2} e3. REVOKE-ACT-FOR-REPLY: *preds*={e1, e2}
`delegate(t, p1, p2)`	e1. DELEGATE-REQUEST: *parms* = {t, p1, p2} e2. DELEGATE (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid, t, p1, p2} e3. DELEGATE-REPLY: *preds*={e1, e2}
`revokeDelegate(t, p1, p2)`	e1. REVOKE-DELEGATE-REQUEST: *parms*={t, p1, p2} e2. REVOKE-DELEGATE (at server): *preds*={e1, *last*-auth-server-update}, *parms*={caller-pid, t, p1, p2} e3. REVOKE-DELEGATE-REPLY: *preds*={e1, e2}

Table A.4: Label Manipulations

Aeolus API Method	Event Records
`addSecrecy(t)`	e1. ADD-SECRECY: *parms*={t}
`removeIntegrity(t)`	e1. REMOVE-INTEGRITY: *parms*={t}
`declassify(t)`	e1. DECLASSIFY: *preds*={*last*-auth-client-update}, *parms*={t}
`endorse(t)`	e1. ENDORSE: *preds*={*last*-auth-client-update}, *parms*={t}

Table A.5: Authority Closures

Aeolus API Method	Event Records
`cid createClosure(class)`	e1. CREATE-CLOSURE-REQUEST: *parms*={class.Name} e2. CREATE-CLOSURE (at server): *preds* = {e1,*last*-auth-server-update}, *parms*={caller-pid, class.Name}, *ret-val*={cid} e3. CREATE-CLOSURE-REPLY: *preds*={e1, e2}, *ret-val*={cid}
`closureObj getClosure(cid)`	e1. GET-CLOSURE: *preds*={*last*-auth-client-update}, *parms*={cid}
`callClosure(cid, closureObj)`	e1. CALL-CLOSURE: *preds*={*last*-auth-client-update}, *parms*={cid, c.pid} e2. CALL-CLOSURE-RETURN *preds*={*last*-event-of-closure-invocation}

Table A.6: Boxes

Aeolus API Method	Event Records
`b createBox(outerSLabel, outerILabel, innerSLabel, innerILabel)`	e1. CREATE-BOX: *parms*={outerSLabel, outerILabel, innerSLabel, innerILabel}
`b.getInnerSLabel()`	e1. GET-INNER-SLABEL: *preds*={b.CREATE-BOX}
`b.getInnerILabel()`	e1. GET-INNER-ILABEL: *preds*={b.CREATE-BOX}
`b.getOuterSLabel()`	e1. GET-OUTER-SLABEL: *preds*={b.CREATE-BOX}
`b.getOuterILabel()`	e1. GET-OUTER-ILABEL: *preds*={b.CREATE-BOX}
`obj b.getContent()`	e1. GET-BOX-CONTENT: *preds*={b.*last*-PUT-BOX-CONTENT/ b.CREATE-BOX}
`b.putContent(obj)`	e1. PUT-BOX-CONTENT: *preds*={b.*last*-PUT-BOX-CONTENT/ b.CREATE-BOX}

Table A.7: Shared State: Shared Objects

Aeolus API Method	Event Records
`obj getSharedRoot()`	e1. GET-SHARED-ROOT *preds*={*last*-UPDATE-SHARED-ROOT}
`updateSharedRoot(obj)`	e1. UPDATE-SHARED-ROOT *preds*={*last*-UPDATE-SHARED-ROOT}
`objID createSharedObject(sLabel, iLabel, obj)`	e1. CREATE-SHARED-OBJECT: *parms*={sLabel, iLabel}, *ret-val*={objID}
`obj getSharedObject(objID)`	e1. GET-SHARED-OBJECT: *preds*={obj.*last*-UPDATE-SHARED-OBJECT}, *parms*={objID}
`updateSharedObject(objID, obj)`	e1. UPDATE-SHARED-OBJECT: *preds*={obj.*last*-UPDATE-SHARED-OBJECT}, *parms*={objID}
`deleteSharedObject(objID)`	e1. DELETE-SHARED-OBJECT: *preds*={obj.*last*-UPDATE-SHARED-OBJECT}, *parms*={objID}

Table A.8: Shared State: Shared Queues

Aeolus API Method	Event Records
`qID createSharedQueue(sLabel, iLabel)`	e1. CREATE-SHARED-QUEUE: *parms*={sLabel, iLabel}, *ret-val*={qID}
`enqueue(qID, obj)`	e1. ENQUEUE: *preds*={q.*last*-ENQUEUE/DEQUEUE}, *parms*={qID}
`obj dequeue(qID)`	e1. DEQUEUE: *preds*={obj.ENQUEUE}, *parms*={qID}
`obj waitAndDequeue(qID)`	e1. WAIT-AND-DEQUEUE: *preds*={obj.ENQUEUE}, *parms*={qID}
`deleteSharedQueue(qID)`	e1. DELETE-SHARED-QUEUE: *preds*={q.*last*-ENQUEUE/DEQUEUE}, *parms* = {qID}

Table A.9: Shared State: Shared Locks

Aeolus API Method	Event Records
`lockID createSharedLock(sLabel, iLabel)`	e1. CREATE-SHARED-LOCK: *parms*={sLabel,iLabel}, *ret-val*={lockID}
`lock(lockID)`	e1. LOCK: *preds*={lock.*last*-UNLOCK/ CREATE-SHARED-LOCK}, *parms*={lockID}
`bool trylock(lockID)`	e1. TRY-LOCK: *preds*={lock.*last*-UNLOCK/CREATE-SHARED-LOCK}, *parms*={lockID}, *ret-val*={bool}
`unlock(lockID)`	e1. UNLOCK: *preds*={lock.*last*-LOCK}, *parms*={lockID}
`deleteSharedLock(lockID)`	e1. DELETE-SHARED-LOCK: *preds*={lock.*last*-LOCK/UNLOCK/ CREATE-SHARED-LOCK}, *parms*={lockID}

Table A.10: File System: Files

Aeolus API Method	Event Records
`bool createFile(hostname, filePath, sLabel, iLabel)`	e1. CREATE-FILE-REQUEST: *parms*={hostname, filePath, sLabel, iLabel} e2. FS-CREATE-FILE (at server): *preds*={e1,*last*-parentdir-update}, *parms*={filePath, sLabel, iLabel} e3. CREATE-FILE-REPLY: *preds*={e1, e2}, *ret-val*={bool}
`content readFile(hostname, filePath)`	e1. READ-FILE-REQUEST: *parms*={hostname, filePath} e2.if(cache-miss):FS-READ-FILE(at server): *preds*={e1,file.*last*-write}, *parms*={filePath} e3. READ-FILE-REPLY: *preds*={e1, e2/cache.file.fs-read}
`writeFile(hostname, filePath, content)`	e1. WRITE-FILE-REQUEST: *parms*={hostname, filePath} e2. FS-WRITE-FILE (at server): *preds*={e1,file.*last*-write}, *parms*={filePath} e3. WRITE-FILE-REPLY: *preds*={e1, e2}
`bool deleteFile(hostname, filePath)`	e1. DELETE-FILE-REQUEST: *parms*={hostname, filePath} e2. FS-DELETE-FILE (at server): *preds*={e1, file.*last*-write, *last*-parentdir-update}, *parms*={filePath} e3. DELETE-FILE-REPLY: *preds*={e1, e2}, *ret-val*={bool}

Table A.11: File System: Directories

Aeolus API Method	Event Records
`bool createDir(hostname, filePath, sLabel, iLabel)`	e1. CREATE-DIR-REQUEST: *parms*={hostname, filePath, sLabel, iLabel} e2. FS-CREATE-DIR (at server): *preds*={e1, *last*-parentdir-update} e3. CREATE-DIR-REPLY: *preds*={e1, e2}, *ret-val*={bool}
`content listDir(hostname, filePath)`	e1. LIST-DIR-REQUEST: *parms* = {hostname, filePath} e2. FS-LIST-DIR (at server): *preds*={e1, *last*-dir-update} e3. LIST-DIR-REPLY: *preds*={e1, e2}
`bool deleteDir(hostname, filePath)`	e1. DELETE-DIR-REQUEST: *parms* = {hostname, filePath} e2. FS-DELETE-DIR (at server): *preds*={e1, *last*-dir-update, *last*-parentdir-update} e3. DELETE-DIR-REPLY: *preds*={e1, e2}, *ret-val*={bool}

Table A.12: File System: File Streams

Aeolus API Method	Event Records
`handle openFilestream(hostname, filePath, mode)`	e1.OPEN-FILESTREAM-REQUEST: *parms*={hostname, filePath, mode} e2.if(mode=$READ$&cache-miss) FS-READ-FILE(at server): *preds*={e1,file.*last*-write}, *parms*={filePath} if(mode=$READ$&cache-hit) cache.file.fs-read if(mode=$WRITE$) no-event e3. OPEN-FILESTREAM-REPLY *preds*={e1, e2}, *ret-val*={handle}
`content readFilestream(handle, offset, count)`	e1.READ-FILESTREAM: *preds*={cache.file.fs-read}, *parms*={handle}
`writeFilestream(handle, content, offset, count)`	e1.WRITE-FILESTREAM: *parms*={handle}
`closeFilestream(handle)`	e1.CLOSE-FILESTREAM-REQUEST: *parms*={handle} if(mode=$WRITE$) e2. FS-WRITE-FILE (at server): *preds*={e1,file.*last*-write}, *parms*={filePath} e3. CLOSE-FILESTREAM-REPLY *preds*={e1, e2}

Table A.13: Service Registry and Remote Procedure Calls

Aeolus API Method	Event Records
`registerService(serviceName, className)`	e1. REGISTER-SERVICE: *parms*={serviceName}
`serviceStub getService(hostname, serviceName)`	e1. GET-SERVICE: *parms*={hostname, serviceName}
`result serviceStub.remoteMethod(params)`	e1. SEND-RPC (at client): *preds*={*last*-auth-client-update}, *parms*={hostname, serviceStub.serviceName, remoteMethod.Name} e2. RECEIVE-RPC (at server): *preds*={e1, service.registration, *last*-auth-client-update}, *parms* = {serviceName, methodName, sLabel, iLabel} e3. SEND-RPC-REPLY: *preds*={last-event-in-method-invocation, *last*-auth-client-update} e4. RECEIVE-RPC-REPLY: *preds*={e1, e3, *last*-auth-client-update}

Table A.14: Local Calls and Forks

Aeolus API Method	Event Records
`fork(pid, code)`	e1. FORK: (in parent thread) *preds*={*last*-auth-client-update}, *parms*={pid} e2. USER-THREAD-LAUNCH (in child thread): *preds*={e1}, *parms*={pid}
`call(pid, code)`	e1. CALL: *preds*={*last*-auth-client-update}, *parms*={pid} e2. CALL-RETURN: *preds*={last-event-in-code-execution}

Table A.15: Node Creation / Deletion

Aeolus API Method	Event Records
`start-up(appName, appArgs, isDataNode)` (this is not part of the API)	e1. CREATE-NODE (at auth-server): *preds*={*last*-auth-server-update}, *parms*={hostname}, *retval*={nodeRootPid} e2. CREATE-NODE: *preds*={e1}, *parms*={hostname, nodeRootPid, isDataNode} e3. LAUNCH-PI: *preds*={e2},*parms*={piID, nodeRootPid, appName} e4. LAUNCH-USER-THREAD: *preds*={e3}, *parms*={nodeRootPid}
`deleteNode(hostname)`	e1. DELETE-NODE-REQUEST: *parms*={hostname} e2. DELETE-NODE (at auth-server): *preds* = {e1},*parms*={hostname} e3. DELETE-NODE-REPLY: *preds* = {e1, e2}

Table A.16: Platform Instance Launch / Shutdown

Aeolus API Method	Event Records
`launchPlatformInstance(hostname, piPid, appName, appArgs)`	e1. PI-LAUNCH-REQUEST: *preds*={*last*-auth-client-update}, *parms*={hostname, piPid, appName} e2. LAUNCH-PI (at hostname): *preds*={e1,pi.node.creation}, *parms*={piID, piPid, appName} e3. LAUNCH-USER-THREAD (at hostname): *preds*={e2}, *parms*={pid=piPid} e4. LAUNCH-PI-RETURN (at hostname): *preds*={last-event-in-app-thread} e5. LAUNCH-PI-REPLY: *preds*={e1, e4}
`shutdownPlatformInstance(hostname, piID)`	e1. PI-SHUTDOWN-REQUEST: *parms*={hostname, piID} e2. SHUTDOWN-PI (at hostname): *preds* = {e1, *last*-auth-client-update}, *parms*={caller-pid, piID} e3. SHUTDOWN-PI-RETURN (at hostname): e4. SHUTDOWN-PI-REPLY: *preds* = {e1, e4}

Table A.17: Auditing Interface

Aeolus API Method	Event Records
`eid createEvent(preds, event-name, parms, status, ret-val, timestamp)`	e1. *event-name*: event info as specified by the arguments

Table A.18: I/O Devices

Type of I/O Event	Event Records
`read`	e1. READ-FROM-IO-DEVICE: *parms*={device-type}
`write`	e1. WRITE-TO-IO-DEVICE: *parms*={device-type}

Appendix B

Implementation Details: Audit Trail Collection and Shipment Structures

This appendix includes a detailed description of the data structures used to collect and ship audit trail data in our current prototype.

B.0.2 Local In-Memory Collection

An Aeolus event is captured by an `EventRecord` object, whose fields store the information associated with the event (the event ID, event predecessors, operation name, parameters, return value, timestamp, and the status). `EventRecord`s are submitted to the platform instance `Collector` for storage. The `Collector` is responsible for aggregating `EventRecord`s from all user threads of this platform instance. Since user threads run concurrently, we need to guarantee that submitting `EventRecord`s to the `Collector` is a *thread-safe* routine. Moreover, since the record data is forwarded periodically to a central server, we also need to synchronize this routine with the removal of the records that have already been shipped. In order to guarantee the thread-safety of these operations without a significant synchronization overhead, we use a specialized concurrent data structure, called `SynchronizedEventsTable`, to

store all the records submitted to the `Collector`.

The `SynchronizedEventsTable` maps user thread IDs to the list of `EventRecords` generated in these threads. Internally, it relies on Java's `ConcurrentHashMap` collection provided by the `java.util.concurrent` package, which is an efficient and scalable thread-safe implementation of `Map` that does not use table-wide locking and is optimized for retrieval operations (*read* operations almost always run concurrently). Since user threads are reused by the platform instance, there is no need to remove any entries from our map - once an event record was submitted to the `Collector` from a user thread, the table will always have an entry for this thread ID; therefore, *write* operations will be very rare, occurring only upon the addition of a new user thread ID to the table. Iterating over map elements is also tread-safe; however, insertions or deletions occurring after the creation of the Iterator may or may not be included (as specified by the Java documentation).

Our `SynchronizedEventsTable` provides two operations: `addEvent` and `getAllAndReset`. The `addEvent` operation adds a new event record to the table based on the ID of the calling user thread: if the ID is already in the table, the event is just appended to the list of events associated with this ID; otherwise, a new table entry is created for this user thread. The `getAllAndReset` operation runs when the event records are being shipped to the central server. It returns all the event records currently stored in the table. In order to retrieve the content of the table, the procedure iterates over all the table entries, assembling the current event lists into a separate collection, while resetting map's events list references to new empty lists (the value of each map entry is actually a special object that holds a reference to the events list). Even though there might be recent additions to the table that were not returned by the map iterator, these will be captured the next time this procedure runs. Both `addEvent` and `getAllAndReset` are *read* operations with respect to the events table, since they only modify the value of the event list referenced from the table entry (either by appending to the existing list or resetting the value to a new list). As already mentioned these operations are optimized by the underlying Java collection and usually require no locking.

Since user threads are single threads, only one `addEvent` operation at a time can run on the same event list and so, we have no race conditions when it comes to appending events to the list. However, the `getAllAndReset` operation can process a table entry concurrently with the `addEvent` operation on that entry's event list. In order to avoid this race condition, we associate a lock with each table entry to synchronize the access to its event list reference. The `getAllAndReset` operation will lock each entry at a time as it iterates over the table. This lock will only be held briefly since the procedure only resets the list reference to a new list. The `addEvent` operation will also need to grab this lock when appending to the list; however, the contention on this lock will be low since `getAllAndReset` only runs periodically when the the data is shipped to the server.

B.0.3 Shipment to a Centralized Location

Audit trail data collected locally by each platform instance is periodically sent to a centralized location for persistent storage and processing. Each platform instance runs a `LoggingClient`, which sends the audit trail data to the central `LoggingService` via Java RMI. When the shipment event happens, the `Collector` passes all the accumulated `EventRecords` to the `LoggingClient`. Scheduling the log shipment is implemented as a recurring task using Java's `ScheduledThreadPoolExecutor` class. This thread pool accepts `Runnable` task objects and can execute them at a fixed time rate with any specified delays using one of its available worker threads. This pool is managed by the platform instance.

In order to facilitate the processing of the received data at the server and reduce the amount of information being sent, all event records are aggregated into a structured form. In our current implementation the records are sent as a Java `HashMap` object, which maps the ID of the user thread to a list of its event records (similar to the form in which the data is locally stored). We rely on Java Serialization to send this map across the network (we also provide an XML alternative for assembling and parsing this data). Along with the event records, each data shipment also includes the ID of the sender platform instance and a system timestamp - all the events in

this shipment will be actually stored with this approximate timestamp to reduce the performance overhead of querying the system clock every time an event occurs.

In our current implementation, the central `LoggingService` runs at one system node. For example, it could be co-located with the authority server.

Bibliography

[1] Neo4j the graph database. *http://neo4j.org.*

[2] D. Banning, G. Ellingwood, C. Franklin, C. Muckenhirn, and D. Price. Auditing of distributed systems. In *Proceedings of the 14th National Computer Security Conference*, pages 59–68, October 1991.

[3] Matt Bishop, Christopher Wee, , and Jeremy Frank. Goal-oriented auditing and logging. *ACM Transactions on Computer Systems*, 1996.

[4] Brent Boyer. Robust java benchmarking, part 1: Issues. *http://www.ibm.com/developerworks/java/library/j-benchmark1.html*, June 2008.

[5] National Computer Security Center. *A Giude to Understanding Audit in Trusted Systems.* July 1987.

[6] Winnie Cheng. *Information Flow for Secure Distributed Applications.* PhD thesis, Massachusetts Institute of Technology, September 2009.

[7] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[8] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

[9] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[11] Teresa F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *Proceedings of the 11th National Computer Security Conference*, pages 65–73, October 1998.

[12] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol.* CRC Press, 2006.

[13] Abdelaziz Mounji, Baudouin Le Charlier, Denis Zampunieris, and Naji Habra. Distributed audit trail analysis. In *In Proceedings of ISOC'95 Symposium on Network and Distributed System Security*, pages 102–112, 1994.

[14] J. Picciotto. The design of an effective auditing subsystem. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 13–22, April 1987.

[15] Katherine E. Price. Host-based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, December 1997.

[16] John Rose. So you want to write a micro-benchmark. *http://wikis.sun.com/display/HotSpotInternals/MicroBenchmarks*, April 2008.

[17] Samuel I. Schaen and Brian W. McKenney. Network auditing: Issues and recommendations. In *Proceedings of the 7th Annual Computer Security Applications Conference*, pages 66–78, December 1991.

[18] Kenneth F. Seiden and Jeffrey P. Melanson. The auditing facility for a VMM security kernel. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 262–277, May 1990.

[19] Stephen E. Smaha. svr4++, a common audit trail interchange format for UNIX. Technical report, Haystack Laboratories, Inc., Austin, Texas, October 1994.

[20] Steen R. Snapp, Stephen E. Smaha, Daniel M. Teal, and Tim Grance. The DIDS (distributed intrusion detection) prototype. In *Proceedings of the Summer 1992* USENIX *Conference*, pages 227–233, June 1992.

[21] Steven Ray Snapp and Stephen E. Smaha. Signature analysis model definition and formalism. In *Proceedings of the 14th Workshop on Computer Security Incident Handling and Response*, August 1992.

[22] M.S. Vaccaro and G.E. Liepins. Detection of anomalous computer session activity. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 280–289, 1982.