Analyzing Audit Trails in the Aeolus Security Platform

by

Aaron Blankstein

S.B., C.S. M.I.T., 2010
S.B., Mathematics M.I.T., 2010

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

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Barbara H. Liskov
Institute Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Analyzing Audit Trails in the Aeolus Security Platform

by

Aaron Blankstein

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation of an analysis system for audit trails generated by Aeolus, a distributed security platform based on information flow control. Previous work focused on collecting these audit trails in the form of event logs. This thesis presents a model for representing these events and a system for analyzing them. In addition to allowing users to issue SQL queries over the audit log, this analysis system provides mechanisms for active monitoring of events. This thesis introduces a new model for event monitoring called *watchers*. These watchers receive updates about events from a watcher manager. This manager allows watchers to specify filters and rules for dynamically modifying those filters. My results show that this analysis system can efficiently process large event logs and manage large sets of queries.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

Acknowledgments

I would like to start by thanking my advisor, Prof. Barbara Liskov. Her help with this project was invaluable. I feel very fortunate to have worked with her and learned from her during my time with PMG. I'm very grateful for all of the patience and effort that she invested in this work and my writing.

I thank everyone in 32-G908 for making my time in the lab more enjoyable and my frequent trips to the coffee machine more social. I would especially like to thank Vicky for the many conversations we spent grappling with the details of the Aeolus platform.

I wouldn't be where I am now if it wasn't for my family. I thank my grandmother for all her support and encouragement over the years. I would like to thank my parents for their support, for worrying about the things I wouldn't, and for putting that first Compaq computer in my room so many years ago. I would like to thank my sister for her care, understanding, and wonderful food recommendations. I'd especially like thank her for being such a tough act to follow.

I would also like to thank my roommate Luke for putting up with me this year and tagging along during both my adventures and misadventures.

I would also like to thank MIT– Anne Hunter, my advisors, professors and friends. I would especially like to thank the Crew team. Finally, I thank the Charles River for everything it has given and taken. Every bend will be missed.

Contents

Chapter 1

Introduction

To manage information securely, a system must be able to detect information misuse. By tracking relevant security events, we can build audit trails that can be used for this purpose. By querying and analyzing these trails, we can track information misuses.

This thesis presents a system to analyze audit trails in Aeolus, a distributed security platform based on information flow control. Aeolus is designed to facilitate the development of secure applications that protect confidential information.

Even in secure systems, malicious administrators, users or developers may be able to launch attacks that tamper with or leak important information. Determining what information was compromised and who was responsible not only enables administrators to stop further abuses, but in many instances has legal ramifications.

Tracking information misuse can be achieved by collecting relevant information into audit trails and then analyzing those trails. By actively monitoring certain processes and events for suspicious activity or violations of high-level policies, applications can be stopped before more violations occur. And by examining past events, the source of violations can be determined.

This thesis presents a model for representing events and an interface for analy-

sis that supports both monitoring events as they occur and selecting events in the past. I provide a design and implementation of this interface that prepares audit trails for querying, accepts query requests, and executes those queries.

1.1 Thesis Outline

The remainder of the thesis is structured as follows. Chapter 2 provides relevant background information on the Aeolus platform and log collection. Chapter 3 presents a user model for representing events and describes the original storage format. Chapter 4 presents an interface for directly querying events. Chapter 5 describes the watcher system for active event monitoring. Chapter 6 describes the various implementation problems involved in creating the user model of events, executing queries, and running watchers. Chapter 7 evaluates the various components of the analysis system in terms of performance. Chapter 8 discusses related work in event tracing systems, streaming databases, and intrusion detection. Chapter 9 presents some topics for future work and reviews the contributions of the thesis. Appendix A details the various attributes of the user event model.

Chapter 2

Aeolus System Overview

My work on analysis of audit information has been carried out on top of a security platform called Aeolus. In this chapter, I present an overview of the Aeolus security model, and relevant details of log collection. More complete descriptions of Aeolus and log collection can be found in Cheng [8] and Popic [18].

2.1 Aeolus System Model

Aeolus is a distributed security platform based on information flow control. In information flow control, all data are tagged with security labels. Security labels are sets of tags, and these tags become associated with processes as they read data. Processes can communicate with other processes within the system and the appropriate tags will flow between them. Processes carrying tags are called contaminated. Contaminated processes cannot write to arbitrary files or communicate outside of the system. Tags can be removed from a process only if that process has authority for that tag.

Aeolus associates each process and piece of information with two labels: secrecy and integrity. As information flows through the system, the following constraints are imposed on the information and the receiver.

$$\text{SECRECY}_{information} \subseteq \text{SECRECY}_{receiver}$$
$$\text{INTEGRITY}_{information} \supseteq \text{INTEGRITY}_{receiver}$$

For example, if a process has secrecy label {SECRET}, it may only write to files containing the tag SECRET in their secrecy label. Integrity, however, must flow towards smaller labels. For example, if a file has integrity label {BOB}, only a process whose integrity label contains BOB can write to the file.

Certain modifications of a process' labels require authority. Declassification, removing a tag from the secrecy label, and endorsement, adding a tag to the integrity label, both require authority. The opposite modifications, however, do not. In Aeolus, authority is derived from principals (users or roles) and processes run on behalf of those principals. Principals gain authority through delegations from other principals.

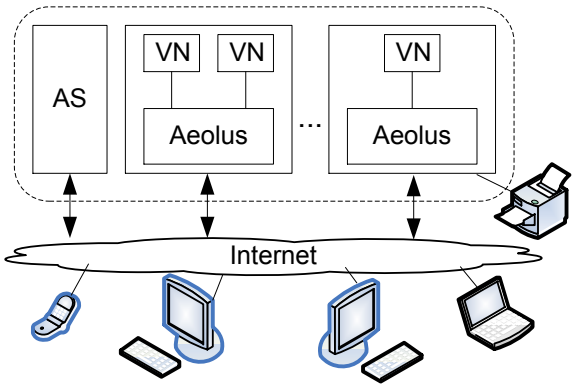


Figure 2-1: High level overview of Aeolus system architecture.

The Aeolus system is designed to support distributed applications. Figure 2-1 shows an overview of the system architecture. Each computer is modeled as a node and each node may run several virtual nodes. These virtual nodes are responsible for running user processes. User processes may communicate with other

processes on the same virtual node using shared state mechanisms and communicate with other virtual nodes using an RPC mechanism. Aeolus manages the flow of labels over shared state and through RPC calls. The Aeolus library and runtime also manages authority state – the collection of all of the tags and authority chains for principals (roles or users) in the system. Additionally, Aeolus provides a file system where each file has integrity and secrecy labels.

2.2 Log Collection

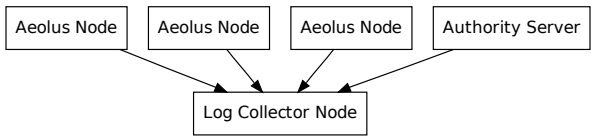


Figure 2-2: Flow of logs to the log collection system.

Aeolus distributes log collection across all nodes in the system. Each node is responsible for logging any events that occur locally. Nodes submit these logs periodically to a central log collector that stores the events for later processing and analysis.

All calls to the Aeolus runtime generate events. For example, when a user declassifies a tag from their label, this creates a Declassify event. Events are ordered to preserve causal relationships. Each event in a single user process holds a dependency to the previous event in that process. Additionally, events may depend on events in other processes or nodes. For example, a Declassify event depends on the last authority update seen by that process.

The format of events is described in detail in Chapter 3.

Chapter 3

Modeling Events

Audit events are represented so that users can write simple queries over simple structures for most analysis tasks. In this section, I present the event model used for querying and active monitoring.

3.1 Event Model

Each event is represented to the user as a data object with a large set of attributes. Events have the following structure:

EventCounter

This is a long integer that is used for determining event dependency. If $e.\mathit{EventCounter} < e'.\mathit{EventCounter}$ then e is certain to not depend on e'.

OpName

This is an integer code representing the name of the operation that generated the event. For example, an event may be a Declassify or Delegate operation. User-specified events have the OpName UserEvent.

OpAttributes

This is a group of attributes storing additional information about the opera-

tion that generated the event. For example, a Declassify event contains OpAttributes for the declassified tag and the result of the declassification (*accepted* or *rejected*).

ContextAttributes

This is a group of attributes storing information about the context of an operation.

3.1.1 Operation Attributes

OpAttributes is a group of attributes storing information specific to the operation that generated an event. These attributes are largely composed of arguments and results from Aeolus library calls. For example, a Fork operation has a field, `SwitchedPrincipal`, that identifies the principal of the forked process.

Different operations store different OpAttributes. For example, file operations contain an attribute `Filename`. A Declassify operation does not have this attribute. Appendix A provides a complete listing of operations and attributes.

3.1.2 Context Attributes

ContextAttributes is a group of attributes storing information about the process and environment of an event. It is composed of the following fields:

Node

This is a long integer that identifies the node that the event ran on.

VirtualNode

This is a long integer identifier for the virtual node of the event. Special virtual nodes, such as the Authority Server, have reserved identifiers.

Process

This is an identifier for the user process that generated the event. System

events such as classloading, occur in a special system process whose identifier is a known constant.

Principal

User processes all run on behalf of a principal. This is an integer identifying that principal.

Secrecy

This is the secrecy label of an event's process at the time it was logged. This is also the secrecy of the event object when it is used within the Aeolus system.

Integrity

This is the integrity label of the event's process at the time it was logged.

Predecessors

This holds a list of `EventCounter` values corresponding to the direct predecessors of this event. All events except the first one depend on the event immediately before them in the same process. Some events also depend on events in a different process. For example, file reads will depend on the last write event to that file. Using these dependencies, the analysis system can guarantee a causal ordering of events.

Timestamp

This attribute stores the local time at the node where an event was collected at the time it was collected. Because local clock times can vary between nodes, this attribute cannot be used to correctly order events.

3.1.3 Information Flow Labels and Context

The secrecy and integrity attributes warrant some additional attention. Because events themselves may convey sensitive information, each event object carries secrecy and integrity labels. For events occurring in a user process, these labels are

the process labels immediately before the event occurred. So, if a process' secrecy label contains a tag *T* before a Declassify event, the secrecy label of the corresponding audit event contains the tag *T*, but labels of subsequent events in that process will not. Events that occur outside of a user process, such as Authority State events, have empty labels.

3.2 Authority Provenance

Answering a simple question about authority related events can become quite difficult. For example, "Who authorized this declassification?" could be answered in a number of ways. If the declassification ran on a process with principal Bob, then Bob is one answer. But if Bob is not the creator of the declassified tag, he must have been granted authority by another user, or be acting on behalf of another user. Answering these queries requires knowing, at the time of the event, how a process became authoritative for a particular tag. This is called the *authority provenance*.

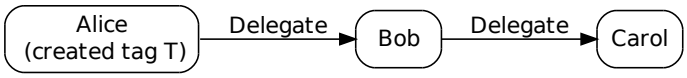


Figure 3-1: This graph represents the authority provenance for Carol's declassification of a tag T. Alice has authority because she created the tag. Alice delegated her authority to Bob. Bob then delegated his authority for the tag to Carol.

The authority provenance is the list of principals that records how a particular principal obtained authority for a particular tag (see Figure 3-1). Declassify and Endorse events contain a field `AuthorityProvenance` that holds the provenance that event used. This field is represented as a list of principals, beginning with the creator of the tag and ending with the running principal.

3.3 Running Principal and the Basis

The Aeolus system allows processes to switch to other principals that they ActFor. If events only provide information about the current principal, this can be used to disguise certain uses of authority. For example, an administrator might be interested in any uses of a principal Bob's authority. Suppose Bob has authority for a tag *ALICE-SECRET* because he has an ActFor delegation from Alice. If Bob switches to principal Alice and performs a Declassify, the event contains no reference to Bob. The authority provenance will only store Alice's provenance and the running principal will point to Alice.

To solve this problem, the running principal field is actually a stack of principals, called the basis, that ends with the currently running principal. This way, when Bob switches to another principal via a Call, the logged event will retain information implicating Bob's involvement.

Chapter 4

Querying Events Directly

In this chapter, I provide the interface for querying past events. The audit trail may be queried using read-only SQL. Events appear in a single table where each event attribute is a distinct column. Events lacking certain attributes hold `NULL` values in those columns.

4.1 Information Flow Constraints

To prevent queries from overly contaminating the issuing process, each query provides secrecy and integrity labels. The querying system must make sure that only events fitting those labels are returned. These events fulfill the following constraints:

$$\begin{aligned} \text{SECRECY}_{event} &\subseteq \text{SECRECY}_{query} \\ \text{INTEGRITY}_{event} &\supseteq \text{INTEGRITY}_{query} \end{aligned}$$

The queries issued against the table see a view of the table containing only those

events matching the provided labels. This technique is called query by label and more detail is in Schultz [21]. Figure 4-1 shows an an example of a few different views of the same event table.

EventCounter	Secrecy
1	{}
2	{alice}
3	{alice, bob}

(a)

EventCounter	Secrecy
1	{}
2	{alice}

(b)

EventCounter	Secrecy
1	{}

(c)

Figure 4-1: Example of returned events for the same query on the same table with different secrecy labels for the query. Figure (a) shows the results with secrecy label $\{alice, bob\}$. Figure (b) shows the results with secrecy label $\{alice\}$ and Figure (c) shows the results with the null secrecy label $\{\}$.

4.2 OpName Constants

The event field OpName stores integers that code for specific operations in the Aeolus System. Rather than having to look up the integer value, however, queries may use the actual operation name. Therefore, queries may be written like the following:

```
SELECT * FROM EVENTS WHERE
 OpName = DECLASSIFY
```

4.3 Special Querying Node

Our system uses a special querying node to accept and execute audit trail queries. Queries are issued via normal Aeolus RPC calls to the method in Figure 4-2.

After running the query, the RPC will return the result set as a QueryResult object. This object has only two (final) fields: *lastEventCounter* and *results*. *lastEventCounter* holds the highest EventCounter value that the database has seen when the

```
QueryResult executeQuery(String query,
                         Label secrecy,
                         Label integrity);
```

Figure 4-2: API for executing direct queries.

query was executed. *results* is a two-dimensional array of Objects storing the result set of the query. The provided parameters *secrecy* and *integrity* are the process labels and are used for query by label as discussed in Section 4.1.

4.4 Direct Querying Examples

A common pattern in Aeolus applications is to delegate specific roles to users. For example, in a medical clinic, each doctor may be represented as a separate principal, but each doctor has been delegated the *doctor role*. A query to determine all the doctors currently seen by the logging system would look like the query in Figure 4-3.

```
SELECT DelegatedPrincipal FROM EVENTS WHERE
 OpName = DELEGATE AND
 DelegatingPrincipal = DoctorRole
```

Figure 4-3: An example query looking for all delegations to DoctorRole.

Another common query is of the form: "What tags has Bob's authority been used to declassify?" This query makes use of the `AuthorityProvenance` field described in Section 3.2. A query for all declassifications using Bob's authority looks like the query in Figure 4-4. That query uses the *=ANY* statement in PostgreSQL [20]. This statement is true if any of the elements of the list is equal to the left operand.

```
SELECT TagRemoved FROM EVENTS WHERE
 OpName = DECLASSIFY AND
 Bob=ANY(AuthorityProvenance)
```

Figure 4-4: An example query using the `AuthorityProvenance` field to find declassifications.

Chapter 5

Watchers and Active Monitoring

The active monitoring system is designed to enable Aeolus users to watch events in the system without excessively burdening them with irrelevant events, requiring excessive authority, or overly limiting their monitoring capabilities. For this task I introduce client objects called *watchers* that receive event notifications from a server called the Watcher Manager. In this chapter I present the design of watchers and the Watcher Manager.

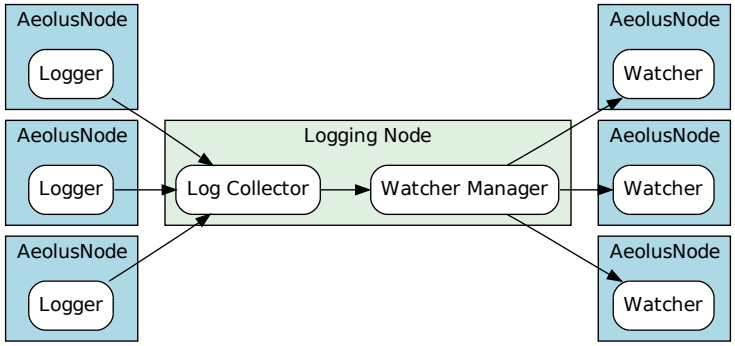


Figure 5-1: Flow of events from originating nodes to watchers.

Clients wishing to receive updates construct a virtual node implementing the watcher interface and register this virtual node with the Watcher Manager. The

Watcher Manager then begins sending events to the watcher. Figure 5-1 shows how events travel from source nodes and ultimately to watchers.

A watcher typically starts by running a direct query to gather some initial information from the audit trail. The watcher then attaches to the Watcher Manager to receive updates. From the direct query, the watcher learns an `EventCounter`. The watcher supplies this `EventCounter` as an argument to ensure that events are not missed between attaching and the initial query.

5.1 Controlling Watcher Contamination

As discussed in Chapter 3, events carry information flow labels. These labels contaminate the watcher as it receives updates. For the watcher to raise an alert, it would need authority for its labels. To prevent itself from becoming overly contaminated, a watcher must specify its labels when it registers. The Watcher Manager will then filter the events that the watcher receives based on those labels.

5.2 Filtering Unneccessary Events

Even while filtering with labels, watchers would receive a large volume of events. Watchers may specify an additional set of filters that reduce the events that they receive.

Each filter represents a set of events of interest to a watcher. At any moment there is a set of filters associated with a watcher and each event is evaluated with respect to those filters. If it matches a filter, it will be sent to the watcher.

5.2.1 Abstract Syntax for Filters

Filters are described using the specification language presented in Figure 5-2. This syntax is defined using an extended BNF grammar. The extension $a\ldots$ denotes a

list of one or more of *a*. Reserved words and terminal symbols appear in boldface font.

```
<filters>              ::= <filter> ...

<filter>               ::= <filter-primary>
                         | <joined-filters>
<joined-filters>       ::= <filter> <conjunc> <filter-primary>
<filter-primary>       ::= <bracketed-filter>
                         | <filter-component>
<bracketed-filter>     ::= ( <filter> )

<filter-component>     ::= <event-field> <operator> <value>

<operator>             ::= CONTAINS | IN | =
<conjunc>              ::= OR | AND
<value>                ::= <atom-value> | <list-value>
<list-value>           ::= [ <atom-value> ... ]
<atom-value>           ::= integer | string | datetime | op-name
```

Figure 5-2: An abstract syntax in for watcher specified filters.

In this syntax, the *event-field* may be any one of the event attributes listed in Appendix A. Figure 5-3 presents an example abstract syntax tree for a watcher's filter set.

Not all filters described by this syntax describe satisfiable events. Section 5.4 describes some additional semantic constraints for the language.

The concrete syntax used in this thesis closely follows the abstract syntax. Watchers submit a specification in a single Java String. This representation is not ideal. Construction of a more suitable concrete syntax is a topic for future work.

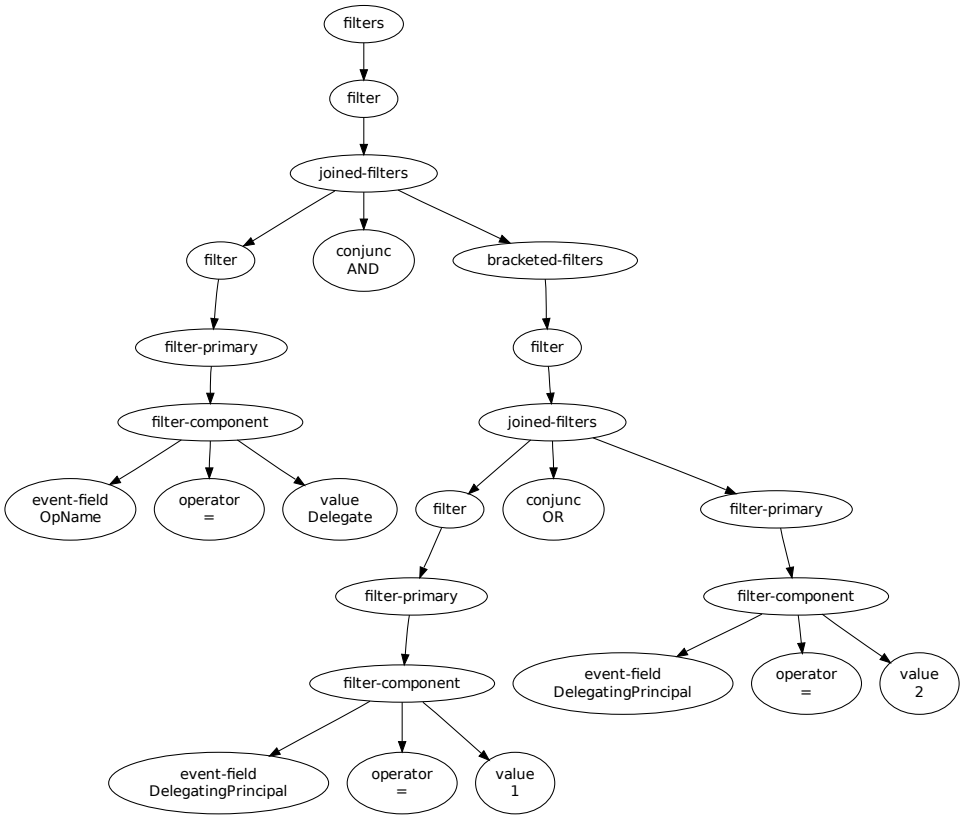


Figure 5-3: This figure shows an example AST for a watcher filter. This filter selects an OpName and any of two DelegatingPrincipals. This tree can be represented with the statement `OpName = DELEGATE` **AND** `(DelegatingPrincipal = 1` **OR** `DelegatingPrincipal = 2)`

5.2.2 Example Filters

An administrator may want to monitor all uses of a particular principal's authority. The watcher can be registered with the following filter (assuming the principal is 1):

```
OpName = DECLASSIFY AND
AuthorityProvenance CONTAINS 1
```

In a medical clinic, administrators may wish to enforce a policy that doctors should not delegate their authority to other users. This cannot be enforced using Aeolus information flow control policies, but can be checked using watchers.

In a simple system, a watcher may be interested in just a few doctors with well-known Principal IDs. The watcher specification in this case is just a single filter:

```
OpName = DELEGATE AND
 (DelegatingPrincipal = 1 OR
  DelegatingPrincipal = 2 OR
  DelegatingPrincipal = 3)
```

The same filter could also be represented using the *IN* operator:

```
OpName = DELEGATE AND
 DelegatingPrincipal IN [1 2 3]
```

In a more complicated system however, new doctors may join the system after the watcher was registered. In this case, a watcher needs a way to modify its filter set. The next section describes how this is done.

5.3 Dynamic Filter Modification

Because the events of interest can change as a watcher runs, a static filter set is not enough. A watcher may additionally specify *rules* that describe these changes.

Each rule provides a filter that describes what events should trigger the change and an action. The action either adds some new filter, or removes filters from the watcher's filter set.

5.3.1 Abstract Syntax for Rules

Rules extend the watcher specification language with the syntax in Figure 5-4. Using this syntax, rules define a *skeletal-filter*. This is similar to a filter except that it allows filter-components to refer to the trigger event that caused the rule to be executed.

```
<rules>                ::= <rule> ...

<rule>                 ::= ON <filter> <action> <skeletal-filter>
<action>               ::= ADD | REMOVE

<skeletal-filter>      ::= <skeletal-primary> | <joined-skeletons>
<joined-skeletons>     ::= <skeletal-filter> <conjunc> <skeletal-primary>
<skeletal-primary>     ::= <bracketed-skeleton> | <skeletal-component>
<bracketed-skeleton>   ::= ( <skeletal-filter> )

<skeletal-component>   ::= <event-field> <operator> $<event-field>
                         | filter-component
```

Figure 5-4: An abstract syntax for watcher specified rules.

5.3.2 Evaluating Rules

Each rule instantiates a filter from the *skeletal-filter* and the trigger event. If the triggered rule is an add rule, the instantiated filter is added to the watcher's filter set. For a remove rule, the instantiated filter is compared against each of the filters in the watcher's filter set and matching filters are removed. A filter matches if it describes a set of events that is a subset of the events that match the rule's instantiated filter.

Filters are instantiated by transforming each *skeletal-component* into a normal *filter-component*. This is done by replacing the *event-field* following $ with the value from the triggering event. For example, if a rule contains the action "OpName = $OpName" and is triggered by a Declassify event, then the rule will instantiate a filter "OpName = Declassify".

5.3.3 Example Rules

In a large medical clinic, doctors are added over the course of a watcher's lifetime. So the watcher must use rules to adapt its filter set to these changes. A particular principal is a doctor if it receives a delegation to the *doctor role*. This role is a well-known Principal ID, which is 10 in the following example. The specification must contain a rule that watches for new doctors and creates a new filter to monitor the delegations of that new doctor, as in the following example:

```
ON
  OpName = Delegate AND DelegatingPrincipal = 10
ADD
  OpName = Delegate  AND
  DelegatingPrincipal = $PrincipalDelegatedTo
```

5.4 Semantic Constraints

Not all instances of the abstract syntax describe valid specifications. To catch these bad specifications, the Watcher Manager imposes several additional semantic constraints on the watcher specification language. Registration will fail with an error if there are unsatisfiable specifications.

For example, a filter fails when an event field is not applicable for a given OpName as in the following filter.

```
OpName = Declassify AND FileName = "/foo/bar/"
```

Event fields also expect certain types for values and if the wrong type is given, registration will fail. For example, the *Principal* field expects an integer. Therefore, the following specification would fail.

```
OpName = Declassify AND Principal = "Foobar"
```

Additionally, event fields should not be specified twice and joined by an **AND**. A simple example is this filter:

```
OpName = Declassify AND OpName = Endorse
```

Semantically bad rules also raise errors during registration.

5.5 Watcher Definition and Registration

To monitor events, an Aeolus application creates a virtual node implementing the interface in Figure 5-5. The Aeolus application then registers with an RPC to the watcher manager.

```
public interface Watcher{
  public void receiveEvents(List<Event> events)
}
```

Figure 5-5: Remote interface for watchers.

```
public void registerWatcher(String watcherSpec,
                            Label secrecy,
                            Label integrity,
                            long startEvent,
                            long delayTolerance)
```

Figure 5-6: RPC method used for watcher registration.

The Watcher Manager runs on a special virtual node with a well-known network address and exposes the *registerWatcher* method (Figure 5-6). The *watcherSpec* parameter is used to supply the rules and filters for the watcher. The *secrecy* and *integrity* specify the labels of the watcher as described in Section 5.1. The parameter *startEvent* is used to specify the event in the past where the watcher should begin receiving events.

After the watcher is registered, the Watcher Manager begins dispatching events by calling the *receiveEvents* method on the watcher. This method accepts a list of

events from the watcher manager. A list is used to prevent excessive communication overhead for watchers receiving large numbers of events. By default, the Watcher Manager collects batches of events for 30 seconds before shipping them to the watcher. In some scenarios, this delay is unacceptable. In these cases, the watcher uses the optional parameter *delayTolerance*. This specifies a maximum delay in milliseconds for the batches. If the delay is set to zero, the behavior is disabled entirely.

Further, the *receiveEvents* method is treated specially by the Aeolus runtime in that only the Watcher Manager virtual node can make calls to this interface. This ensures that fraudulent events are not received.

Watchers are implemented in Java and therefore can respond to incoming events as needed.

5.6 Ordering Guarantees

The Watcher Manager provides several guarantees for the ordering of *receiveEvents* calls.

1. All events matching a watcher's filter set will be sent to the watcher.
2. Changes from rules are applied after the triggering event and before the next event passes through the Watcher Manager.
3. The list of events is ordered by *EventCounter*.
4. All the events in a call to *receiveEvents* occur after the last *EventCounter* of the previous call.
5. *receiveEvents* will only be called after the previous call has completed.

These guarantees prevent possible synchronization issues for the watcher.

The optional parameter *startEvent* specifies the point in the audit trail where the watcher should attach. The watcher will receive all the events that match its filter set and have a higher *EventCounter* than *startEvent*.

5.7 Example Watcher Registration

The previous doctor monitoring examples in this chapter assume that all doctors in the system are created after the watcher registers. Of course, this may not be true. Capturing all the doctors in the system requires mixing direct querying and active monitoring. Example code for querying information and registering this watcher is presented in Figure 5-7.

```
1  String getDRs =
2      "SELECT DelegatedPrincipal FROM EVENTS " +
3      "WHERE OpName = DELEGATE AND " +
4      "DelegatingPrincipal = " + DoctorRole;
5
6  String wsl =
7      "OpName = DELEGATE and (?), " +
8      "ON OpName = DELEGATE and DelegatingPrincipal ="
           +
9      DoctorRole + " ADD"
10     "OpName = DELEGATE  and " +
11     "DelegatingPrincipal = \$PrincipalDelegatedTo";
12
13 QueryResult drResult = executeQuery(getDRs,
14                                     Label.empty,
15                                     Label.empty);
16 String startingDoctors = "";
17 boolean first = true;
18 for(Object[] row : drResult.results){
19   if(first){
20     startingDoctors += "DelegatingPrincipal = ";
21     first = false;
22   }else{
23     startingDoctors += " or DelegatingPrincipal = ";
24   }
25   startingDoctors += row[0];
26 }
27
28 registerWatcher(String.format(wsl, startingDoctors),
29                 Label.empty, Label.empty,
30                 drResult.lastEventCounter);
```

Figure 5-7: Pseudocode for using a direct query to gather information prior to watcher registration. The unbound variable *DoctorRole* is assumed to be a well known principal. Lines 13-15 gather all the doctors currently in the logging system. Lines 16-26 are format the result to insert it into the watcher specification. Lines 28-30 register the watcher. Notice that the registration needs to use *drResult.lastEventCounter* to ensure that it does not miss any events between the query and the registration.

Chapter 6

Implementation

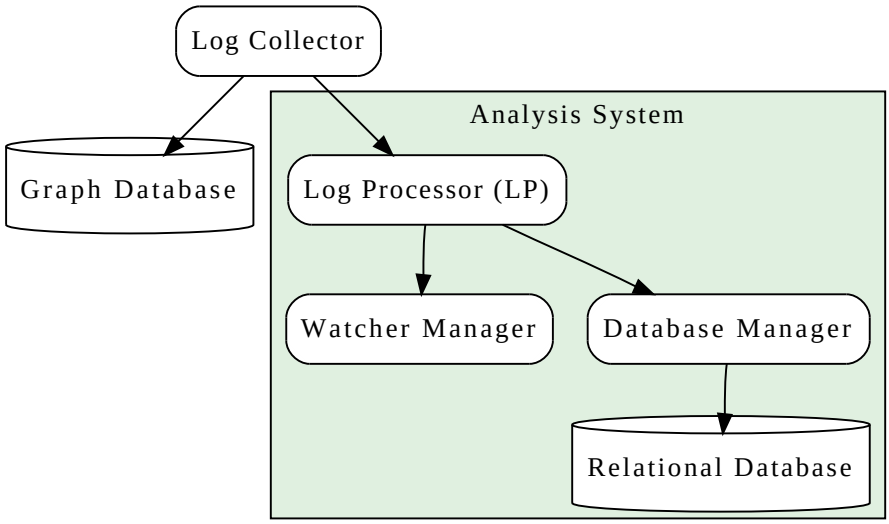


Figure 6-1: This is an overview of the various components involved in preparing events for querying and handling active monitoring.

The complete implementation of the analysis system is composed of over 5,000 lines of Java code, and an ANTLR grammar.

Figure 6-1 presents a high-level overview of the analysis system components. The Log Collector forwards events to the Log Processor (LP). The LP formats the events and enforces a proper ordering before shipping events to the Database Manager and the Watcher Manager. The Database Manager stores the events in a rela-

tional database and the Watcher Manager ships events to watchers.

The Log Collector additionally stores events in a graph database, Neo4J [16]. This is used for producing filtered views of audit data in Popic [18].

This rest of this chapter describes the implementation of the LP, the Database Manager, and the Watcher Manager components.

6.1 Log Processor

6.1.1 Collected Event Format

During log collection, events are collected and stored using a format that differs from the user model discussed in Chapter 3. Each event is represented as a tuple:

```
<EventID, Operation, Arguments, ReturnValue, Timestamp,
  Predecessors>
```

This form differs from what was described previously in two ways:

- The collected events contain ordering information that differs from the `EventCounter`. Each event has an Event ID, and events identify other events that directly precede them by including their IDs in the Predecessors field. For example, the event recording the start of executing an RPC would contain the ID of the event recording the sending of that RPC request in this field.

- The context of an event is mostly missing, in order to keep the overhead low during log collection. Instead, the context for an event e can be computed by looking at information in the predecessor events of e.

The LP must perform three tasks before events are ready for either direct querying or watchers:

1. The EventCounter field of the event model needs to be calculated.

2. Events shipped to users must contain explicit context information, such as the running principal or the process labels. This must be constructed by traversing events in the audit trail.

3. Direct querying requires building a second representation of the logging data, a relational database.

6.1.2 Ordering Events

The LP receives events from the Log Collector in a queue. This queue is ordered by the time an event is received at the Log Collector and does not respect the ordering imposed by the Predecessor relationship. The LP, however, must process events only after the events they depend on.

When new events are received, the LP checks each of its predecessors. If all predecessors have already been processed, the LP processes the current event. Otherwise it adds the event to a table for later processing. When the LP finishes processing an event, it checks if any events are waiting on it and tries to process them again.

As part of processing an event, the LP assigns it an `EventCounter`, which it computes by incrementing a counter. This event count is certain to be greater than that assigned to any events that were predecessors of the current event because the LP processes events only when they are ready based on their list of Predecessors. This way the LP ensures that the total order provided by the `EventCounter` is consistent with the causal order determined by the predecessors. Popic [18] contains a detailed discussion of this issue.

6.1.3 Adding Context Information

Each user process has associated security information, such as the authority principal it runs on behalf of, or the current secrecy and integrity labels. This information is not stored for each event in the user process. Instead, the LP calculates it by using information in events that preceded the current event.

To gather information about an event e, the LP examines the immediate predecessors of that event. Because the LP processes events in a causal order, the predecessors of e already contain complete context information. The LP derives the context of e by taking the context of its predecessor and applying any modifications caused by the predecessor. For example, if e is preceded by a Declassify event d, the context of e is the same as the context of d except with a tag removed from the secrecy label.

The LP needs a way to quickly find predecessor events. To do this, it keeps a map from Event ID to the event objects.

Some events do not have predecessors in the same process. For example, the VirtualNodeStart event has no such predecessor. The LP identifies these events and constructs the initial context information for a process.

6.1.4 Tracking Authority Provenance through Cache Lines

When an application attempts to declassify (remove a tag from its secrecy label), the Aeolus system performs an authority check. This check involves finding an authority provenance for the running principal. This provenance is logged with the Declassify or Endorse event. However, the Aeolus system caches authority checks so that subsequent checks do not have to do a full provenance calculation. In this case, the authority provenance is not readily available to the logged event.

To solve this problem, each node adds a CacheWrite event whenever a cache line is written. This event holds the authority provenance and a cache line identi-

fier. Authority related events all depend on the most recent CacheWrite event for the cache line they use. When the LP sees a CacheWrite, it creates a mapping from the `EventCounter` to the authority provenance. Then, as it sees events referencing that CacheWrite, it fills in the provenance.

6.2 Database Manager

My system supports fast analysis of past events by direct querying of the event log. Events are stored in a relational database, and indices make most queries efficient. The Database Manager (DBM) is responsible for storing events in the relational database. In this section, I outline the implementation of this component.

6.2.1 Storing Events with Different Sets of Attributes

As I discussed in Chapter 3, different event types have different sets of attributes. There are several ways to deal with this problem.

1. Store additional event attributes in separate tables
2. Store different event types in different tables
3. Store all events in a wide table with nullable columns

The first and second approach minimize table width and do not require a large number of null fields. To support the types of queries described in Chapter 4, the Database Manager would have to modify incoming queries to do the correct JOIN and UNION operations. However, I chose to store all the events in the same table.

A concern with using a wide table for storage is space usage. However, PostgreSQL uses an efficient representation for storing null values that limits the overhead of null fields [19]. Null fields also affect the performance of indexes in PostgreSQL, because null values won't be indexed. Queries involving IS NULL oper-

ators will not be able to use indexes. However, my event model does not require the use of IS NULL in normal circumstances.

6.2.2 Information Flow Labels and Events

The direct querying interface from Chapter 4 presents a table view based on information flow constraints. The implementation of this relies on the PostgreSQL Information Flow Control (PGIFC) database provided by David Schultz [21]. This is a modified version of the PostgreSQL database that supports label operations.

The DBM inserts events into the database and explicitly sets the labels of those events. This requires that the DBM pass an additional parameter to the underlying database.

The DBM also deals with incoming queries. When a query comes in, the DBM executes the query on a read-only connection. Additionally, it provides a parameter that filters the output based on the supplied labels.

6.2.3 Table Indices

Table indices are an important part of providing an efficient querying structure. The following fields are indexed:

- `EventCounter`
- `OpName`
- `FileName`
- `DelegatingPrincipal`
- `DelegatedPrincipal`
- `RunningPrincipal`

Because the database is only used for INSERT and SELECT statements, additional indices do not significantly decrease INSERT performance. Depending on different use cases, additional indices could be added.

6.3 Watcher Manager

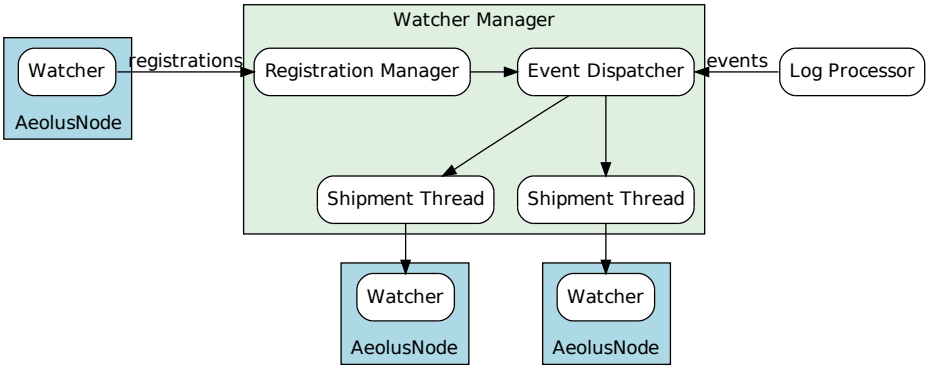


Figure 6-2: This figure provides an overview of the Watcher Manager.

The Watcher Manager is responsible for handling watcher registrations, evaluating rules and filters, and dispatching events to the watchers. Figure 6-2 provides an overview of the various components involved in the Watcher Manager. Watchers send registration requests to the Registration Manager. This component performs some semantic processing before forwarding the request to the Event Dispatcher. The Event Dispatcher receives events from the LP and dispatches those events to appropriate shipment threads. Each watcher has a shipment thread that handles calling the receiveEvents method on the watcher.

This section examines the implementation of the three Watcher Manager components: the Registration Manager, the Event Dispatcher, and the Shipment Threads.

6.3.1 Registration Manager

Adding a new watcher to the manager requires parsing the given filters and rules and performing some semantic processing before passing the request onto the Event Dispatcher.

```
public interface Watcher{
  public void receiveEvents(List<Event> events)
}
```

Figure 6-3: Remote interface for watchers.

The Registration Manager receives *registerWatcher* RPC's (see Figure 6-3). It then uses a parser constructed with ANTLR [17] to generate an abstract syntax tree (AST) for the provided filters and rules. This AST is a simple Java representation of the input specification. The Registration Manager then does semantic processing on this AST and checks it for any bad filter or rule definitions. Finally, it attempts to forward the registration information to the dispatcher. If this process fails at any point, the *registerWatcher* RPC raises an exception and the watcher is notified that registration has failed.

Semantic Language Processing

The Registration Manager processes watcher specifications to fit a canonical form. This form is used to simplify semantic checks and the job of the Event Dispatcher. In this section, I describe this form and provide some of the reasoning for each added constraint.

First, filters with *OR* statements are rewritten into multiple patterns. Without *OR*s the dispatcher never needs to backtrack when evaluating filters or checking remove rules. This also allows the next processing step, sorting filter components.

Second, filter components are sorted by the *event-field* they are matching. This makes finding specific filter types efficient. For dispatching, this allows the dispatcher to quickly determine what operation name a particular filter is interested in. This also makes remove rules faster to evaluate, as comparing sorted patterns can be done very quickly.

Finally, redundant filter components are removed so that excess checks are avoided at dispatch and rule evaluation.

6.3.2 Event Dispatcher

The Event Dispatcher maintains a table of filters and rules for all the currently active watchers. It uses this table to evaluate rules and dispatch events to appropriate shipment threads.

The Dispatcher serially examines events coming from the LP in a two step process. In the first step, the event is evaluated against the active filters. If a filter matches, the Dispatcher adds the event to the correct Shipment Thread.

In the second step, the Dispatcher evaluates the event against the active rules. If the event causes a rule to fire, that rule's action is evaluated and the Dispatcher modifies the active filter table appropriately.

Accepting New Watchers

The Registration Manager forwards watcher registrations to the Event Dispatcher. Registration requests provide a *StartEvent* that specifies where in the event log the watcher should begin receiving events. The Event Dispatcher has to deal with the fact that this is in the past.

To do this, the Event Dispatcher manages a buffer of events from the past 30 seconds. If *StartEvent* is further in the past, the Dispatcher raises an error that propagates back to the Registration Manager and to the watcher.

If *StartEvent* is inside the buffer, the Dispatcher stops processing new events while it catches up the new watcher. The Dispatcher performs the normal two step event dispatching process except that it only checks one watcher. Once the watcher is caught up, the Dispatcher resumes normal operation.

Because this behavior slows down the Event Dispatcher, the actual implementation starts a special thread to handle a new watcher. This thread evaluates filters and rules the same as the normal dispatcher. Once this thread is within some small distance of the current event in the dispatcher, the dispatcher stalls while the watcher finishes getting to the latest event. Finally, the watcher is added to the

dispatcher, the thread is discarded, and normal dispatching resumes.

Fast Matching for Common Filters

It is important that the Event Dispatcher scale well with many filters. A naive strategy is to apply each event to every filter and rule. This performs reasonably well with a small number of filters, but performance can degrade quickly.

To mitigate some of the cost of additional filters, I use a hash table to quickly select the filters monitoring events with a particular `OpName`. In the worst case, many filters can all be watching the same `OpName`. However, the system will still have significant performance improvements for events without that `OpName`. Section 7.2 presents the performance impact of this strategy.

6.3.3 Automatic Removal of Unsatisfiable Patterns

Under special circumstances, the dispatcher can determine that certain active filters are unsatisfiable. For example, when a dispatcher witnesses a process termination event, any filters looking for that process are unsatisfiable. When this happens, the dispatcher will automatically remove the filter.

This process can be achieved without too much overhead by adding hash tables to map from a process number to filters monitoring that process. Then, when a process termination event passes through the dispatcher, it checks that hash table for filters to remove. A similar technique is used for monitoring the shutdown of virtual nodes, the removal of shared state objects, and file deletions.

6.3.4 Shipment Threads

The watcher model requires that events be shipped to watchers in the order they occur and that each call to *receiveEvents* complete before the next call. The Shipment Threads are responsible for making these calls.

Depending on the implementation of particular watchers, this RPC call could take a long time to complete. Using one thread per watcher prevents watchers from delaying shipments to each other.

The dispatcher queues events for these threads using concurrency-safe queues. As new events appear in its queue, the thread sends the events to the watcher.

Chapter 7

Performance Evaluation

In this chapter I present the results of three experiments to evaluate the performance of the analysis system. The first experiment examines the performance of the Log Processor (LP). Next I present an evaluation of the Event Dispatcher. Finally, I present the total added latency of the active monitoring system.

I performed these evaluations on a computer with an Intel Q9550 CPU with four cores clocked at 2.83 GHz, 4 GB of physical memory, and a 7200 RPM hard drive. The analysis system ran with a 64-bit OpenJDK JVM (build 19.0-b09) on the 2.6.31-22 Linux kernel.

7.1 Log Processor

In this section, I examine some of the performance characteristics of the LP. The throughput of the log processor is measured as the average number of events passing through the processor per second. The log processor is evaluated using a queue of events that resides in memory when the evaluation begins.

The LP is an important bottleneck in the system because it is a sequential process and therefore a hard limit for scaling the system. This system should be able to handle reasonably large systems, though. For example, the Massachusetts General

Hospital system handles about 4,000 patient visits a day [11]. If we overestimate the number of events by assuming each of these visits generated 100,000 logged events, the LP would have to deal with about 5,000 events per a second.

The results in Figure 7-1 show that a single-threaded LP is capable of processing 150-200 thousand events per second. For most systems, this should be enough. I discuss possible strategies for scaling the LP to multiple processors or machines in Section 9.2.

The benchmark fuzz-0 is composed of 100,000 CreateTag, AddSecrecy, and Declassify events. The event log for this benchmark contains a large number of multiple dependency events – the CreateTag and Declassify events have predecessors in the user process and predecessors at the Authority Server.

10% of the events in the medical-bench benchmark are Call events. This benchmark provides a more realistic estimate of an actual workload. Additional work is done in comparison to the fuzz-0 benchmark because Call events require some additional processing to calculate a new running principal. Figure 7-1 shows the decreased throughput.

Benchmark	Throughput (events/second)
fuzz-0	266,180.8
medical-bench	170,622.2

Figure 7-1: Comparison of Processor throughput on several test logs.

7.2 Dispatching Quickly

In this section, I evaluate the Event Dispatcher component and compare the fast dispatch implementation to a naive implementation where all filters are checked against all events.

I expect the Event Dispatcher to scale to large numbers of filters with minimal performance degradation. Scaling is important because large systems could de-

fine thousands of filters. In the medical clinic example (Section 5.7), a watcher that monitored each doctor in the system would require one filter per doctor. In Massachusetts General Hospital, there were over 1,900 clinical staff and 3,700 nurses in 2009[11]. MGH would require over 5,000 filters for just this watcher.

The system is evaluated with watchers running locally to avoid network overhead. All of the filters provided to the system are identical and the event log is entirely loaded into physical memory. The provided filter is `OpName = AddSecrecy AND TagAdded = 30`. I examine two separate workloads with 100,000 events each. 1% of the events in the first benchmark are AddSecrecy events. This increases to 9% in the second benchmark.

I expect real workloads to look more like the first benchmark– watchers are looking for evidence of misbehavior and so few events should ever match.

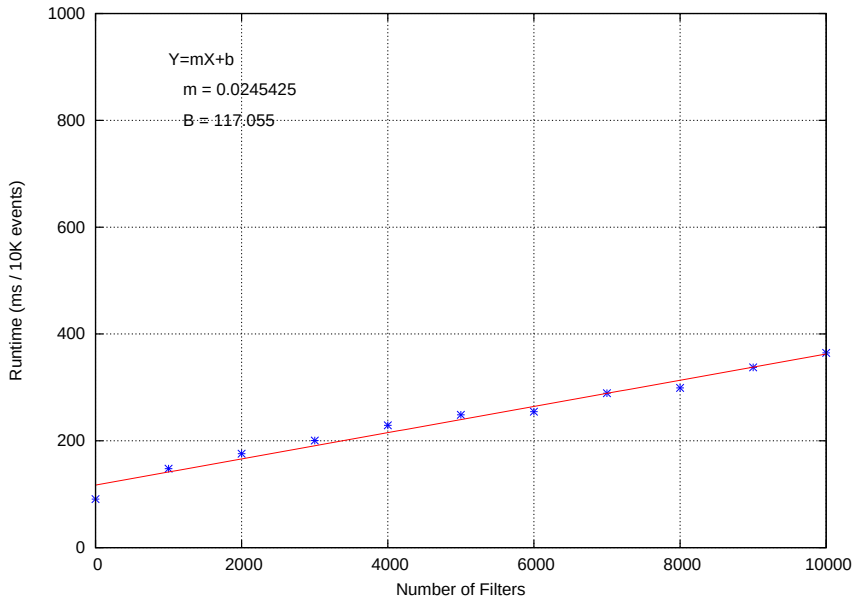
Figure 7-2 compares the performance of a naive dispatcher implementation and an implementation that dispatches based on the `OpName` field. In this evaluation, 1% of the events in the input log are AddSecrecy events. This figure shows that both dispatchers scale linearly with the number of the filters. However, the fast dispatcher experiences much less deterioration. The best-fit slopes show each additional filter only costs about 1% of what it does for the naive dispatcher.

Figure 7-3 presents a worst-case scenario for the Event Dispatcher. In this scenario, 9% of the events in the log are AddSecrecy events. The fast dispatcher still scales better than the naive dispatcher, in this case about 15 times better. This figure also shows that the naive dispatcher scales more poorly than it did with the first benchmark. This happens because the naive dispatcher only has to evaluate the second filter component of the filter when the event is an AddSecrecy event. Because there are more AddSecrecy events, each additional filter costs more to evaluate.

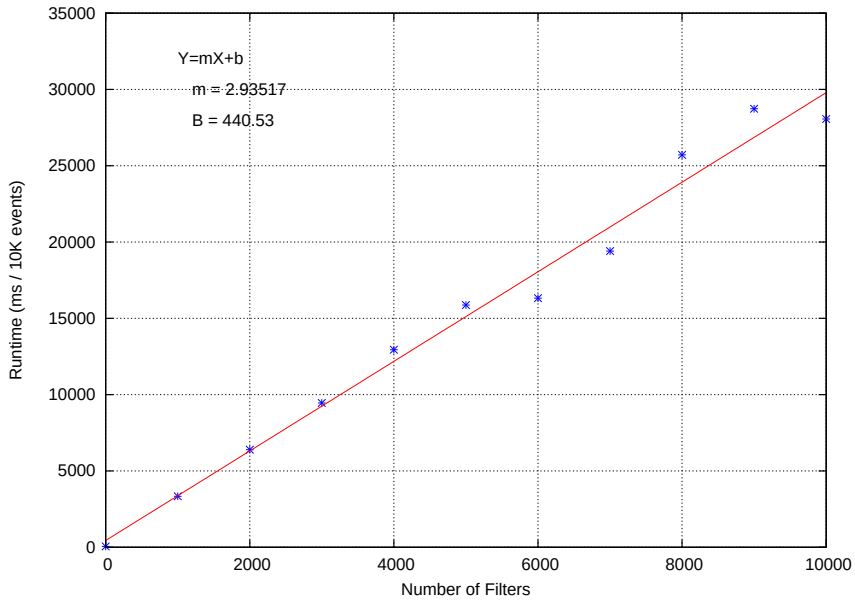
Importantly, the slope for the fast dispatcher is about 9 times higher than in Figure 7-2. This happens because there are about 9 times as many partial matches

as before, so each additional filter costs about 9 times as much.

The fast dispatcher performs well in the first benchmark. At 5,000 filters, it takes 248 milliseconds to process 10,000 events. This is a throughput of about 40,000 events per second, which is good for large systems. However, in the second benchmark, the dispatcher only achieves throughput of 7,800 events per second. Though I expect workloads to be more similar to the first benchmark, this is still a problem. However, the Event Dispatcher's job, unlike the LP's job, exhibits a great deal of parallelism. Implementing a concurrent Event Dispatcher could lead to significant performance gains. I discuss another possible strategy in Section 9.2.

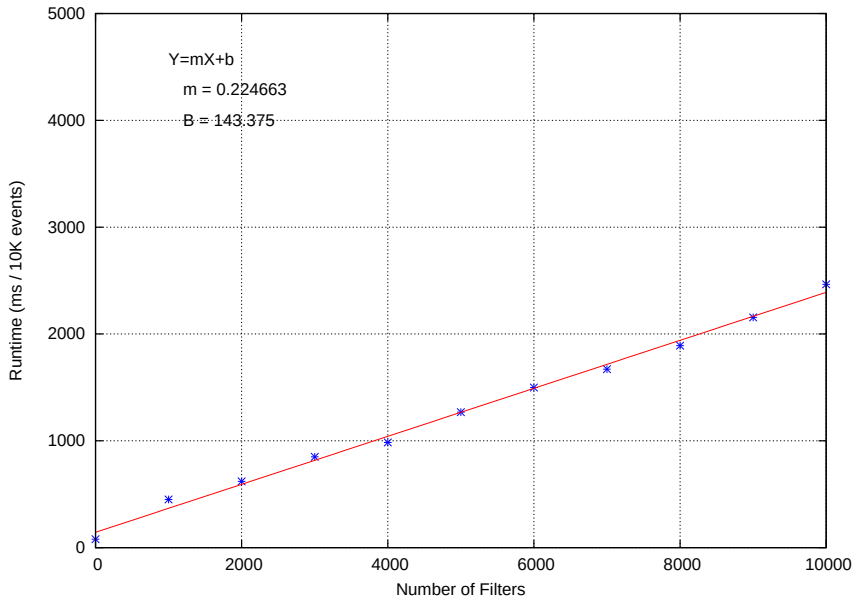


(a) Fast Dispatch

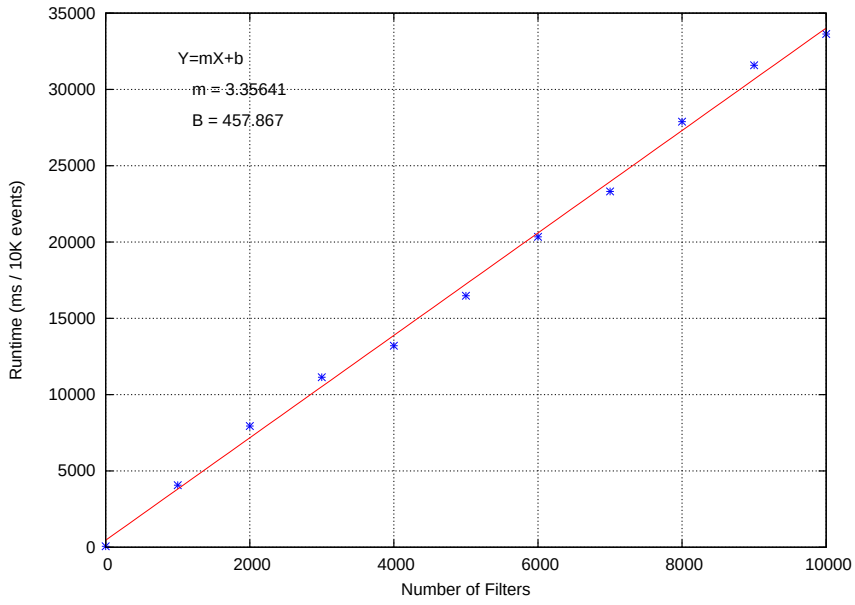


(b) Naive Dispatch

Figure 7-2: Performance comparison of dispatchers as the number of active filters varies. In this example, the filters match only 1% of the events in the log.



(a) Fast Dispatch



(b) Naive Dispatch

Figure 7-3: A comparison of dispatchers as the number of active filters varies. In this example, the filters match about 9% of the events in the event log.

7.3 Event Detection Latency

The last evaluation I provide for the watcher manager system is the combined latency of the log processor and the dispatcher. If the time an event is dispatched to a dispatcher thread is t_d, and the time an event entered the log processor's queue is t_s, then the latency of the event is $t_d - t_s$.

The table in Figure 7-4 provides the average and maximum latency of events on several benchmark logs. Each benchmark runs with 100 active filters, each of which match 10% of the supplied event log.

This data shows that the watcher manager does not add significant latency to event analysis. Since Shipment Threads hold batches of events for up to 30 seconds before shipping them to watchers. In light of these variables, if the added latency of the watcher manager system is on the order of milliseconds, then its effect is negligible.

Benchmark	Average Latency (ms)	Maximum Latency (ms)
fuzz-0	3.75	28
medical-bench	1.95	14

Figure 7-4: Latency measurements for events moving through the watcher manager on several test logs.

Chapter 8

Related Work

In this chapter I discuss some related work from the fields of event tracing, stream databases, and intrusion detection.

Event Tracing and Dynamic Instrumentation Systems

Active log monitoring is closely related to work in tracing and dynamic instrumentation because both require interfaces for specifying events to monitor and actions to take. There are many dynamic instrumentation and tracing systems such as Pin [12], AspectJ [13], and DynamoRIO [6], but I will focus on DTrace [7].

The DTrace system allows users to monitor system calls on a single machine. The authors present a domain-specific language to describe what events should be monitored and if any processing should be done. This language is particularly relevant to my system though DTrace differs in some key aspects that affect interface and language design. First, DTrace is designed to monitor events on a single machine, while Aeolus is a distributed platform. This requires that users be able to specify somewhat more complex patterns for matching events. Second, DTrace inserts monitoring code directly into system calls. This requires that monitoring code have simple event filters and strictly defined memory behavior. In audit trail

monitoring, on the other hand, applications monitor events as they are added at the logging server. The performance and behavior of filters, then, does not affect the applications they are monitoring, just the audit trail server. This allows the system to employ more complex filtering behavior without disrupting application performance. Finally, for DTrace to collect information from the environment when a system call is made, users must provide actions – snippets of code – that specify how to gather information. In Aeolus, system events are logged automatically with all of the information relevant to that event. In some cases, applications may wish to log additional events. Applications only need to supply actions in these cases.

DTrace provides users with useful library functions to perform typical event analysis tasks, such as collecting the average of some value or summing execution times. My system provides no such library. Instead, users write plain Java code. Determining useful functions and libraries for audit trail analysis is a topic for future work.

Streaming Databases

Systems such as Cayuga [9] [10] and SASE+ [3] use streaming databases for active filtering. These systems provide a different model from the filters and rules model presented in this thesis. First, these systems constrain the time between related events, but watchers are typically interested in events that are very far apart. For example, a doctor may misuse his authority months after he becomes a doctor. Second, watchers need to see every event that matches at the time that it matches. Cayuga and SASE+ wait to collect all events that match before shipping them to the monitor. A final point is that our system requires a way to remove filters that are no longer needed, e.g., when a doctor leaves the clinic, we no longer need to monitor his actions. Cayuga and SASE+ use time to stop filtering events; we

instead need to recognize events that cause filters to be removed.

However, systems like Cayuga and SASE+ provide more power than we do. We assume that all computation with events is done in the watcher. By contrast these systems allow much of this processing to be specified via queries. For example if we were monitoring activity on an ATM card, the watcher would need to see all events concerning use of that card, whereas with the streaming database approaches, the watcher could be notified only if these events happened too frequently within some time period. Extending the watcher system to allow some processing to occur at the watcher manager is a possible topic for future work.

In systems such as Aurora [2] and Borealis [1], clients provide queries in the form of data flow processors. Borealis presents a model similar to the filters and rules in this thesis. In Borealis, filters are defined using data flow processors and the user may specify triggers that dynamically modify these processors. However, this system is designed to handle complex queries involving various computational components. Because of this, they provide weak ordering guarantees during dynamic changes. For example, when an event triggers a dynamic change, later events may be seen before that change takes effect. In this case, those events will be missed. Borealis and my active monitoring system share one major feature, though: the ability to begin monitoring events at a time in the past.

The streaming databases XFilter [4] provides another querying model. This system defines a query language where users can specify filters over XML. Candidate XML is compared against the filter using graph matching techniques. If the Aeolus event log is represented as a graph, graph matching techniques could be used to filter events. In this case, a user could derive some of the context information of an event from graph matching queries. For example, a user could filter events based on the principal basis by defining a graph matching query that searched for principal switches. However, these queries could become excessively costly as relationships between events could form chains with lengths in the tens of

thousands. These types of queries work in XFilter because XML typically defines shallow graphs.

Active log monitoring does share some implementation details with XFilter, though. For example, both XFilter and my monitoring system optimize dispatches for particular data attributes, in my case OpNames. This is the mechanism I call *fast dispatching* in Section 6.3.2.

Intrusion Detection

There has been some significant recent work in using audit trails for intrusion detection and recovery. RETRO [14] uses audit trails to repair a compromised system after an intrusion. BackTracker [15] analyzes operating system events to determine the source of a given intrusion on a single system. The events in these system are represented as a DAG where each edge represents a causal relationship. Aeolus treats audit trails similarly. However, the analysis performed in BackTracker is very different from active monitoring or direct querying. While my analysis tools are more focused on discovering misuses or violations, BackTracker is concerned with discovering *how* a misuse occurred.

When an intrusion is discovered, BackTracker traverses the event DAG backwards from that discovery point searching for the source of the intrusion. Because my system analyzes events occurring across a distributed application, backtracking graph analysis is more difficult– audit logs may branch significantly. For example, a user may want to trace the source of a write to a file. The analysis starts by examining the process that wrote to the file. Any inputs to that process could have contributed to the write. These inputs could be file reads, remote procedure calls, accesses to shared state, or startup parameters. Those inputs require further analysis of what processes may have affected them. While this thesis does not explore backtracking analysis over the audit trail, this is a good topic for future work.

Chapter 9

Future Work and Conclusions

This chapter summarizes the contributions of this thesis and presents some possible directions for future work in audit trail analysis.

9.1 Contributions

This thesis has presented the design and implementation of an analysis system for audit trails collected by the Aeolus security platform. I presented a model for representing events in the system, a method for querying those events, and a new model for actively monitoring those events.

This thesis makes the following contributions:

1. I define a usable model for events logged by the Aeolus security platform.

2. I present a method for determining the sensitivity of logging information. This method prevents information leakage through audit trails while using existing authority and information flow control enforcement.

3. I introduce the concept of watchers. This is a new model for active monitoring of an audit trail that allows users to perform complex analysis while limiting communication overhead.

4. I implemented and tested a full analysis system for the Aeolus security platform. This system enables queries over past events and active event monitoring. It achieves high throughput and low latency for active monitoring tasks.

9.2 Future Work

There are several possible topics for continuing work in analysis of Aeolus audit trails. In particular, archiving logged events, building better language tools for active monitoring, scaling log processing, and examining additional optimizations for the Event Dispatcher are all areas of interest.

Archiving event logs is a necessary task. Large systems could generate hundreds of thousands of events a day. In such an environment, direct queries will quickly become infeasible. Further, storage will be extremely costly. However, removing or archiving old events presents several problems. The primary problem is that some old information will need to be available to new direct queries. Differentiating between important events and unimportant events is difficult. For example, a query for all users having authority for a particular tag should always return the complete set, but a query that asks for all uses of some particular principal's authority may not necessarily need to return old events.

This thesis presents a language for describing filters and rules that is too cumbersome in practice. Users must construct Java Strings using results from direct queries. The analysis system could instead provide a similar interface to JDBC Prepared Statements [5], which allow users to specify a query using a combination of a String definition and other Java Objects. The system could also allow users to imbed direct queries into the specification itself. Further, as I discussed in Chapter 8, the system could also provide a library for common analysis functions.

There are a number of issues with respect to the scaling of log processing to

multiple processors or machines. Importantly, the Log Processor requires that an event be processed after the events it depends on. The LP could potentially be adapted to run on multiple CPU cores if events were distributed amongst the cores to minimize cross-core dependencies. Running the LP in a distributed environment poses a larger challenge. The cost of sharing information and synchronizing between different machines could be too burdensome. Properly partitioning events may be able solve this problem.

The Event Dispatcher implemented for this thesis only optimizes dispatches based on `OpName`. When many filters use the same `OpName`, the performance of the dispatcher degrades. An implementation that dynamically constructs indices when large numbers of filters share common features could achieve better performance.

Appendix A

Event Attributes and Context Information

A.1 Operations

- Endorse
- Declassify
- RemoveIntegrity
- AddSecrecy
- Call
- Fork
- VirtualNodeStop
- VirtualNodeStart
- LoadClass
- SendRPC
- AcceptRPC
- RPCReply
- RegisterRPC
- Delegate
- ActFor
- RevokeDelegate
- RevokeActFor
- CreatePrincipal
- CreateTag
- CreateSharedObj
- ReadSharedObj
- WriteSharedObj
- RemoveSharedObj
- CreateFile
- ReadFile
- WriteFile
- RemoveFile
- ListDirectory
- CreateDirectory
- RemoveDirectory
- AppEvent

A.2 OpAttributes

Attribute	Valid Operations
TagAdded	AddSecrecy, Endorse
TagRemoved	RemoveIntegrity, Declassify
AuthorityProvenance	Endorse, Declassify
DelegatingPrincipal	Delegate, DelegateRevoke, ActFor, ActForRevoke
DelegatedPrincipal	Delegate, DelegateRevoke, ActFor, ActForRevoke
TagDelegated	Delegate, DelegateRevoke
SwitchedPrincipal	Call, Fork
Hostname	SendRPC, VirtualNodeStart, VirtualNodeStop
Classname	LoadClass, RegisterRPC, Call, Fork
MergeIntegrity	RPCReply, AcceptRPC, CallReply
MergeSecrecy	RPCReply, AcceptRPC, CallReply
ExtraInformation	AppEvent
CallerPrincipal	AcceptRPC
Filename	ReadFile, WriteFile, CreateFile, RemoveFile, ListDirectory, CreateDirectory RemoveDirectory
ObjectSecrecy	ReadFile, WriteFile, CreateFile, RemoveFile, ListDirectory, CreateDirectory RemoveDirectory
ObjectIntegrity	ReadFile, WriteFile, CreateFile, RemoveFile, ListDirectory, CreateDirectory RemoveDirectory

A.3 General Attributes and Context Fields

- `EventCounter`
- `OpName`
- `Node`
- `VirtualNode`
- `Process`
- `Principal`
- `PrincipalBasis`
- `Secrecy`
- `Integrity`
- `Predecessors`
- `Timestamp`

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM. Available from: `http://doi.acm.org/10.1145/1376616.1376634`, `doi:http://doi.acm.org/10.1145/1376616.1376634`.

[4] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[5] Lance Andersen. Jdbc 4.0 specification. Technical Report JSR 221, Sun Microsystems, Inc., 2006. Available from: `http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html`.

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. Available from: `http://portal.acm.org/citation.cfm?id=776261.776290`.

[7] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[8] Winnie Wing-Yee Cheng. *Information Flow for Secure Distributed Applications*. Ph.D., MIT, Cambridge, MA, USA, August 2009. Also as Technical Report MIT-CSAIL-TR-2009-040.

[9] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Extending Database Technology*, EDBT'06, pages 627–644, 2006.

[10] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research*, pages 412–422, 2007.

[11] Massachusetts General Hospital. Massachusetts general hospital annual report 2009. Available from: `http://www.massgeneral.org/assets/pdf/AR2009lr.pdf`.

[12] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM Press, 2005.

[13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. Available from: `http://portal.acm.org/citation.cfm?id=646158.680006`.

[14] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association. Available from: `http://portal.acm.org/citation.cfm?id=1924943.1924950`.

[15] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23:51–76, February 2005.

[16] Neo4j. Available from: `http://neo4j.org`.

[17] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[18] Victoria Popic. Audit trails in the Aeolus distributed security platform. Master's thesis, MIT, Cambridge, MA, USA, September 2010. Also as Technical Report MIT-CSAIL-TR-2010-048.

[19] PostgreSQL. *Database Physical Storage: Database Page Layout*. Available from: `http://www.postgresql.org/docs/9.0/interactive/storage-page-layout.html`.

[20] PostgreSQL Development Team. *PostgreSQL 9.0.4*. PostgreSQL Global Development Group, 2010. Available from: `http://www.postgresql.org`.

[21] David A. Schultz and Barbara H. Liskov. Ifdb: Database support for decentralized information flow control. In *New England Database Day*, NEDBDay '11, 2011.