Summarizing Audit Trails in the Aeolus Security Platform

by

Wissam Jarjoui

S.B., C.S M.I.T., 2011

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 2012

Author .....
Department of Electrical Engineering and Computer Science
August 15, 2012

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

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

Summarizing Audit Trails in the Aeolus Security Platform

by

Wissam Jarjoui

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Aeolus is a programming platform that supports the development of secure applications that preserve the confidentiality of information entrusted to them. An important part of the Aeolus platform is an auditing subsystem that maintains a log in which it stores information about every security related event that occurs while applications run. The log allows later analysis to determine whether the security policies of the application have been followed.

For an Aeolus user, analyzing an Aeolus event log can prove to be a daunting task, especially when this log grows to include millions of records. Similarly, storing such an event log can be very costly. The system I present in this thesis provides an interface that allows the creation of user-defined summaries of the Aeolus audit trails, as well as marking of events in the log for future archiving or deletion. Our system makes it easier to analyze the Aeolus event log and less costly to store events of interest. This is done through the use of a *QuerySystem* and *SummaryObjects*. I present the system in the context of a sample application based on the financial management service *www.mint.com*. The system is an extension to the Aeolus library; it is implemented in Java code and uses PostgreSQL 9.0 as its primary database.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Barbara Liskov, for the mentorship she gave me during my two years here at PMG. Her guidance helped me navigate and learn a lot about systems security. I am truly grateful for the understanding, patience and time investment she put in while helping me see this project through; I hope that one day I would be able to reflect those traits as a leader.

A big thank you to my colleague, David Schultz, for his tremendous help in this project and my personal development; from suggesting and brainstorming ideas to improve my project, to explaining technical concepts, and being a strong source of knowledge and inspiration and a good role model in general. David, it has been a pleasure working with you.

I would also like to thank James Cowling, Dan Ports and Barzan Mozafari and everyone in 32-G908 for all the fun conversations and useful advice they provided. I am very fortunate to have had the chance to hear and learn from their experiences, and be inspired by them.

As this thesis culminates five years for me at MIT, I would like to take this chance thank my friends and family. My friends, for being their for me, and for offering me help even when I didn't ask for it. Their presence was a home away from home, and their support guided me in times of uncertainty. My family, without whom I would not be where I am today, for their support and encouragement, for looking out for me, and for setting the bar high. In one way or another, I will always have something to learn from them.

Finally, I would like to thank MIT for all what it has given me, in both computer science and real life. I thank Anne Hunter, my advisors, my professors and my TAs for holding my hand when I took on different challenges. I thank the Stata Center, the Green Building, and The Infinite Corridor for being the monuments of my experience here at MIT, and I thank the Charles River, for being the source of calmness in a very busy time. They will all be missed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Maintaining the security of confidential online information, such as medical records and financial data, is a very important task. Recent research has focused on Decentralized Information Flow Control (DIFC) as the most promising approach to enable application developers to secure information. DIFC is based on the principle that the system tracks information as it flows through the system, and only allows information to be released if the releaser has sufficient authority. In fact, DIFC allows for fine-grained control of information flow so that security policies can be tailored to the needs of individual users and organizations.

This thesis extends the security support provided by the Aeolus platform. Aeolus is a platform that combines DIFC and an intuitive security model framework to make it more convenient for developers to build secure applications on a distributed system. Additionally, Aeolus provides automatic auditing of every security related event that occurs while an application runs. Furthermore, it provides a way for applications to log additional events that are meaningful at the application level. The audit trail is an important component of overall security which allows for the discovery of errors that cause security policies to be subverted; the audit trail can also be used to discover attacks.

A common issue in most auditing systems, including the Aeolus system, is that the audit trails can become extremely large. This is especially true if the system being audited is large, e.g., has millions of users, and is long-lived. Therefore, a way

of reducing the stored information and making the important information more easily accessible is needed.

This thesis addresses this problem. It provides a framework that allows groups of events in the audit trail to be summarized: the important content of the group of events is captured in a *summary event*, which is much smaller than the group. Once information has been summarized, the base events that underlie the summary can be moved to archival storage, or even deleted. An additional benefit is that summary events make it easier to access application-specific information, than the base events they summarize, because they can be defined to explain the information in application-specific terms.

A final point is that the production of the summary events is itself controlled by information flow. This is important because, everything a system does is detailed in the audit log, and therefore, there is a potential for information leaks or data corruption if the log could be accessed by an unauthorized party.

The next two sections describe the motivation and outline of this thesis.

1.1 Motivation for Summarization and Archiving

Summarization allows developers to group information that is spread out over the Aeolus audit trails into smaller space. This allows for future archiving or truncation of the Aeolus log, as well as possibly allowing for faster queries to be used.

Take for example a bank's web administrator who is interested in detecting suspicious activity on customers' bank accounts. One way to do this is to produce a plot of the number of outbound transfers carried through a user's account for every week in the last year. Any spikes in the rate of outbound transactions per week could mean that a user's account has been compromised by an attacker. Finally, the administrator might wish to periodically produce such a plot.

Producing such information is possible through the current auditing mechanisms available in Aeolus. However, in order to accomplish such a task, the administrator's audit will have to scan all events in the system in the past year, which could span

millions of events, on a periodic basis, and this could be very costly in terms of time and resources.

Summarization solves this problem by allowing the web administrator to store periodic computations as summaries themselves, and simply reusing these summaries in future audits. For example, the web administrator could store the average number of outbound transfers a user has made per week in the last year and use that as a benchmark for future user activity. Furthermore, summarization supports the archiving and deletion of events by providing the administrator with a way to ensure that important information is not lost in the process.

1.2 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the Aeolus security system. Chapter 3 presents a sample Aeolus application based on the financial management service *www.mint.com*. Chapter 4 describes an example of application-level logging and possible uses of the audit trails. Chapter 5 describes the summarization models and interfaces. Chapter 6 describes examples of how to use our summarization system. Chapter 7 describes the implementation details of the system. Chapter 8 discusses related work. Chapter 9 reviews the contributions of this thesis and presents some topics for future work.

Chapter 2

Aeolus System Overview

This chapter presents an overview of the Aeolus security platform, on which our summarization system is based, and highlights relevant details such as log collection. More complete descriptions of the Aeolus security platform and its log collection and analysis can be found in Cheng [5], Popic [12] and Blankstein [3].

2.1 System Architecture

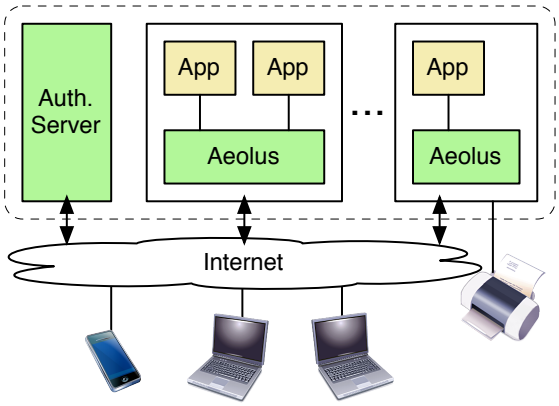

Figure 2-1: Aeolus System Architecture. High level overview of Aeolus system architecture.

The Aeolus architecture is shown in figure 2-1. The system consists of many nodes, each of which is trusted to enforce Aeolus information flow rules.

Aeolus tracks information flow within each system node and between system

nodes. Nodes in the system communicate via RPC messages, those messages are encrypted to protect their secrecy and integrity. Nodes outside the system are considered to be untrusted: information can flow outside of the system only if it is uncontaminated, and information arriving from the outside is marked as having no integrity.

Threads in Aeolus run on behalf of *principals*. The ability of a thread to carry out *privileged operations* is determined by the *authority* of the principal it runs on behalf of. Aeolus tracks the authority of principals in the *authority state* stored at the authority server (AS). Principals, privileged operations and authority are described in sections 2.2.1, 2.2.2 and 2.2.3, respectively.

2.2 Information Flow Model

This section describes the basic concepts and rules of the Aeolus security model.

2.2.1 Principals, Tags and Labels

Aeolus employs an intuitive security model to implement information flow control. The model revolves around three key concepts: *principals*, *tags* and *labels* [4]. Principals represent entities in the system that create, modify and share information. Tags represent security categories of information. A principal authoritative for a certain tag can modify or share information categorized by that tag. Labels are sets of tags and are used to determine whether information can flow from a source to a destination based on the information flow rules described in the following section.

2.2.2 Information Flow Rules

Aeolus allows information to flow from a source S to a destination D only if the following rules are satisfied:

$$SECRECY_S \subseteq SECRECY_D$$

$$INTEGRITY_S \supseteq INTEGRITY_D$$

Threads in an Aeolus node run with security and integrity labels associated with them; objects such as files also have such labels. In Aeolus, a thread cannot modify labels of objects; however, the labels of threads can change, and hence a thread would have to modify its own labels in order to read or write data.

Certain label manipulations, called *privileged manipulations* are unsafe because they remove constraints on information flow:

1. *Declassfication* Remove a tag from a secrecy label.

2. *Endorsement* Add a tag to an integrity label.

A thread is allowed to carry out privileged label manipulations if it is running on behalf of a principal that has authority for the tags affected by these manipulations.

2.2.3 Authority

Authority determines whether a thread can perform privileged label manipulations. Authority starts with tag creation: when a thread creates a tag, its principal has authority for that tag.

Subsequently, the Aeolus authority state can be modified through *grant*, *act-for* or *revoke* operations. Grant operations allow a principal to delegate authority for a particular tag to another principal. *Act-for* operations allow a principal to delegate all of its authority to another principal. Revoke operations remove *act-for* and *grant* links between principals. To avoid covert channels, Aeolus permits only threads with null secrecy labels to modify the authority state.

2.2.4 Compound Tags

Applications frequently have sets of tags that are closely related. In order to simplify the authority structure of the application, Aeolus allows for tags to be grouped upon

creation using *compound tags*. For example, in a medical clinic, patient data tags are *subtags* of an *ALL-PATIENT-DATA* tag, a *supertag*.

A principal authoritative for a supertag is also authoritative for all of its subtags. Similarly, having a supertag in a thread's secrecy label is equivalent to having all subtags in its secrecy label. This reduces label size substantially, and makes label manipulations involving paricular groups of tags less expensive.

See figure 3-1 on page 29 for an example of an authority state model, detailing *act-for*, *grant* and *subtag* relationships in the financial services application described in chapter 3.

2.3 Programming Model

This section explains the programming abstractions Aeolus provides, and how they support DIFC.

2.3.1 Threads and Virtual Nodes

Virtual nodes (VNs), shown as applications in figure 2-1, communicate via RPC and have many threads inside them. A VN is given a principal when it is created and all threads run with this principal or some principal it acts for. An RPC is run in its own threads with the VN principal but the labels of the caller; on the return the labels are sent back to the caller where they are merged with those of the caller: a merge is a union of the secrecy labels and an intersection of the integrity labels. Then the caller continues with its own principal.

2.3.2 Shared State Objects and Boxes

Within a virtual node, threads can share state via special Aeolus shared objects. Aeolus gives threads access to a *root* shared memory object, with null labels.

Aeolus shared obects have immutable labels, and Aeolus ensures that the rules in section 2.2.2 are respected. Users can create their own shared objects; Aeolus ensures

that label manipulations do not take place inside the object's methods.

2.3.3 Authority Closures and Reduced Authority Calls

Aeolus provides developers with *authority closures*. An authority closure is an object bound to a principal at the moment of creation. Threads can later call the closure's methods, which run with the closure's principal. Authority closures allow threads to process confidential information without being exposed to the information itself.

Aeolus also provides a mechanism for threads to reduce their authority, called *reduced authority calls*. To make a reduced authority call, a thread specifies a function and a principal to run it with. The calling thread's principal must act for the principal of the reduced authority call.

With those two mechanisms in place, Aeolus makes it possible to ensure that each part of the application runs with only the authority it needs. Aeolus also provides a principal that is not authoritative for any tags, P_{PUBLIC}, for developers to use when they need to ensure that a part of a program cannot leak any information.

2.3.4 Files

Aeolus provides a network file system that enforces DIFC. Similarly to shared memory objects, files have immutable labels and the rules in 2.2.2 apply for information flow in and out of them.

Files are yet another way (RPCs, shared memory objects) through which Aeolus allows threads to communicate. A complete description of the Aeolus file system API is presented in McKee [10].

2.3.5 Log Collection

Aeolus provides automatic auditing of every security related event that occurs while an application runs; it also provides a way for applications to log additional events that are meaningful at the application level.

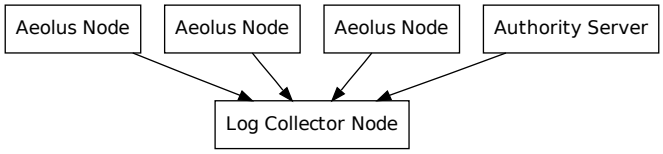

Figure 2-2: Aeolus Log Collection. Flow of logs to the log collection system.

Aeolus distributes log collection across all nodes in the system. VNs send events that occur locally to the log collector as shown in figure 2-2. The log collector stores the events for later processing and analysis, as explained in [3].

Aeolus provides application-level logging to allow developers to log their own events using the following interface:

```
AeolusLib.createEvent(String app_op, List<String> args)
```

This creates a new record in the Aeolus event log. The *app_op* and *args* argument are stored as attributes for that record, allowing the developer to query the event log based on the arguments they provided. The *app_op* parameter is used to to identify the application operation name, and the *args* paramter allows the developer to specify any additional information to store for the event.

Aeolus Event Attributes

Aeolus stores different attributes for different events that take place in the system. Blankstein [3] provides a complete description of those attributes; here we will highlight the ones more relevant to the use of summarization:

event_counter

The event counter uniquely identifies an event, and provides an ordering for the occurrence of the event, i.e. events with a higher event_counter took place after events with a lower event_counter.

timestamp

This is the real time at which the event happened.

secrecy

This is the secrecy label of the thread which caused the event, just before the event took place.

integrity

This is the integrity label of the thread which caused the event, just before the event took place.

op_name

This field specifies the type of the event, e.g. a *DECLASSIFY* event, or a *SENDRPC* event. Application-level events created with the above events all have the *APPLICATION_LVL* operation name.

app_op

This field holds the name of the application-specific operation and is provided by the developer as shown in the interface above (the first argument).

app_args

This field holds the arguments specified for an application-level event, also provided by the developer as shown in the interface above (the second argument).

running_principal

This field holds the principal of the thread that caused the event.

It is important to note that the *secrecy* and *integrity* attributes are important to prevent information leakage in the system: a thread can only read those event records if the DIFC rules in section 2.2.2 are satisfied.

Chapter 3

Mint

Prior to the development of our summarization system, it was important to develop a sense for what is needed from such a system, and what problems can it solve. To answer these questions, I developed an Aeolus application based on the financial management service *mint.com*; from here on we will refer to this application as the "Mint application" or "our application". The *Mint* application served as a case study and guided the development of the summarization system.

In this chapter I present the Mint application model and implementation, setting the stage for later chapters that discuss logging and summarization for this application.

3.1 Mint Model

Mint.com is an financial management service that provides its users with the ability to monitor their bank accounts across different banks from one place. It also allows users to run transaction analysis tools that will access the information of certain transactions for each bank for the user, and generate a result. In addition, Mint runs aggregate analysis for all user accounts.

For example, Bob could sign up on our application and add his Bank of America account and his US Bank account for monitoring. This will cause our Mint application to store credentials for both of these banks for Bob. Bob can then request a graph that

presents his expenditures on food in the last week, based on credit card transactions retrieved from his banks. Bob's usage of our application would also affect the general aggregate analysis done over all Mint users.

This example gives us a starting idea of how should we build our authority structure. For instance, there should be a *BOB-DATA* tag associated with Bob's information (e.g. expenditure statistics, transactions, etc.) that only a principal running on Bob's behalf can declassify. Note that Bob's password should never be revealed to Bob himself, neither should any of Bob's bank credential be revealed as well. Furthermore, there's a natural grouping of user data tags, and hence a super tag, *ALL-USER-DATA*, could be useful to run the aggregate analysis tools for Mint.

We can also see the need to use Aeolus' shared memory objects to store user session information, and use Aeolus' file system library to store persistent information on disk. Other important Aeolus' abstractions will come in handy as well, for example closures for authentication as well as reduced authority calls for processing Bob's history of transactions.

In this section we examine the security model of the application more closely.

3.1.1 Authority State Model

In this section we identify the prinicpals running in the system, the tags associated with different data, and the relationships between them. Figure 3-1 shows an overview of the authority state model for Mint.

An intuitive way to list all necessary[1] principals, is to think of the different "clearance levels" in the system: some information should be accessible by a user, some by a bank, and some information should not be accessible at all. In the light of these requirements, the application employs the following principals:

Mint principal

This principal is authoritative for all tags in the system, and is used for aggregate analysis of user data and user authentication.

[1]Neccessary by good design. The application could run with just one principal.

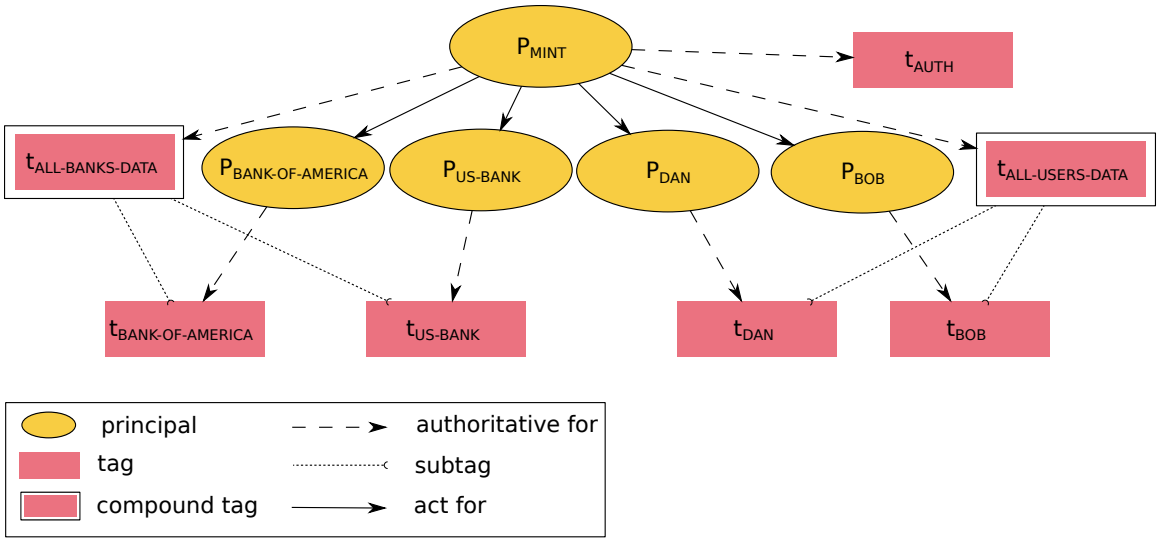

Figure 3-1: Mint Authority State Model. This diagram illustrates the authority state model of the Mint application in an example where two banks are registered at the application, Bank of America and US Bank. Users Bob and Dan have both signed up. Principals are denoted $P_{<name>}$ and tags are denoted $t_{<name>}$.

Bank principals

Each of these principals is authoritative for a particular bank's tag.

User principals

Each of these principals is authoritative for a particular user's tag.

Public principal

Described in section 2.3.3, this principal is used when no privileged operations are necessary to ensure no information leaks.

Similarly, to separate data in the system according to its purpose, (e.g. user password, user bank credentials, etc..), the application employs the following tags:

User data tags

Each of these tags is associated with data carrying a particular user's information. For example, history of a user's transactions. Reading (or modifying) user information requires declassifying (or endorsing) a user data tag.

Bank data tags

Each of these tags is associated with data carrying a particular bank's infor-

mation. Reading a user's bank credentials requires declassifying a bank data tag. This tag is important so that only a bank's closure (a closure bound to the bank's principal) is able to access a user's bank credentials for authentication.

All-Users-Data tag

This tag is a super tag for all the user tags. Running aggregate analysis on Mint user accounts requires declassifying this tag.

All-Banks-Data tag

This tag is a super tag for all the bank tags. Adding a new bank to Mint requires endorsing this tag.

Authentication tag

This tag is associated with user's Mint passwords. Authenticating a user's Mint credentials requires declassifying this tag. This tag is important to prevent release of a user's password to someone who might have gained access to their account.

3.1.2 Files

The application uses Aeolus files for persistent storage. Figure 3-2 shows the hierarchy of the Mint files and their labels, described below:

mint-dir

This is the root directory for the Mint application. The secrecy label is empty, the integrity label contains the *ALL-USERS-DATA* and the *ALL-BANKS-DATA* tags.

banks-dir

Contains a *bank-info* file for each bank that was registered to the application. This directory has an empty secrecy label, and has the *ALL-BANKS-DATA* tag in its integrity labels. Each of the *bank-info* files in this directory has a *BANK-DATA* tag in both its secrecy and integrity labels.

Files	Secrecy Label	Integrity Label
mint-dir	{}	{$t_{ALL\text{-}USERS\text{-}DATA}$, $t_{ALL\text{-}BANKS\text{-}DATA}$}
banks-dir	{}	{$t_{ALL\text{-}BANKS\text{-}DATA}$}
Bank of America-info	{$t_{BANK\text{-}OF\text{-}AMERICA}$}	{$t_{BANK\text{-}OF\text{-}AMERICA}$}
US Bank-info	{$t_{US\text{-}BANK}$}	{$t_{US\text{-}BANK}$}
users-dir	{$t_{ALL\text{-}USERS\text{-}DATA}$}	{$t_{ALL\text{-}USERS\text{-}DATA}$}
Bob	{t_{BOB}}	{t_{BOB}}
bob-info	{t_{BOB}}	{t_{BOB}}
bob-password	{t_{AUTH}}	{t_{BOB}}
bob-bankofamerica-creds	{t_{BOB}, $t_{BANK\text{-}OF\text{-}AMERICA}$}	{t_{BOB}}
bob-usbank-creds	{t_{BOB}, $t_{US\text{-}BANK}$}	{t_{BOB}}
Dan	{t_{DAN}}	{t_{DAN}}
dan-info	{t_{DAN}}	{t_{DAN}}
dan-password	{t_{AUTH}}	{t_{DAN}}
dan-usbank-creds	{t_{DAN}, $t_{US\text{-}BANK}$}	{t_{DAN}}

Figure 3-2: Mint File System Hierarchy. In continuation of the example in figure 3-1, Bob added credentials for Bank of America and US Bank to his account, and Dan added credentials for US BANK to his account.Tags are shown under the secrecy label and integrity label columns, and are denoted $t_{<name>}$. t_{BOB} and t_{DAN} are both user data tags. $t_{BANK-OF-AMERICA}$ and $t_{US-BANK}$ are both bank data tags. t_{AUTH} is the authentication tag, and $t_{ALL-USERS-DATA}$ tag is a supertag as shown in figure 3-1.

users-dir

Has the *ALL-USER-DATA* tag in both its secrecy and integrity labels. Contains a subdirectory per user, each subdirectory contains a *USER-DATA* tag in its secrecy label, and a *USER-DATA* tag in its integrity label. Each of these subdirectories contains the following files:

user-info

Contains the user tag for that user. This file has the *USER-DATA* tag in both its secrecy and integrity labels.

user-password

Contains the user's password. This file contains the *AUTH* tag in its secrecy label, and the *USER-DATA* tag in its integrity label.

user-bank-creds

Contains user credentials for a particular bank. The *users-dir* directory contains one *user-bank-creds* file per bank the user added to their

account. This file has the *BANK-DATA* and *USER-DATA* tags in its secrecy label, and the *USER-DATA* tag in its integrity label.

3.1.3 Shared Memory Objects

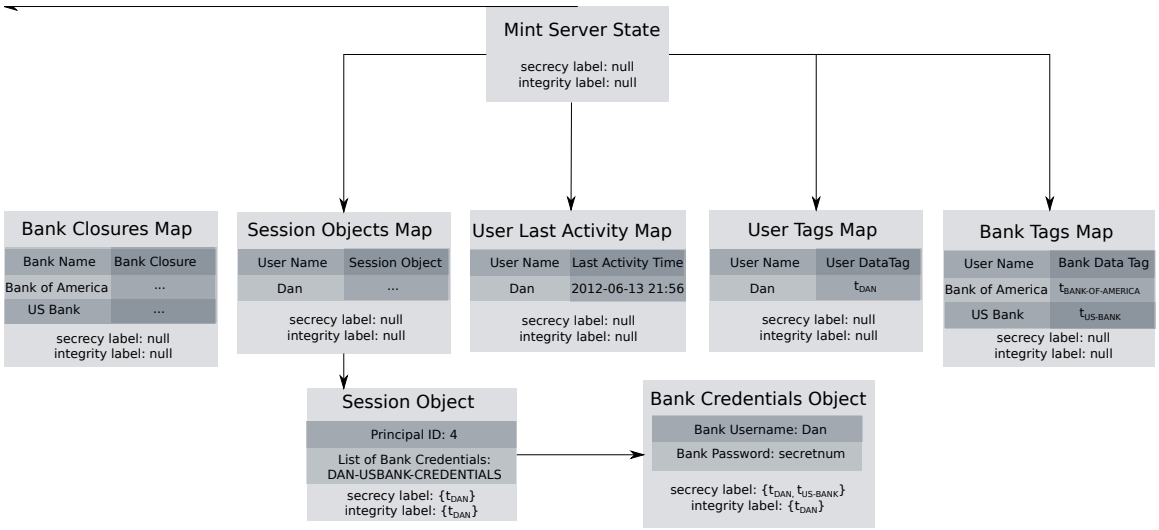

Figure 3-3: Mint Shared State Objects. This figure illustrates what shared memory objects the Mint application uses. This diagram builds on the example presented in Figure 3-2. Bob's session had timed out, and no session object or last activity time is stored for his username. Meanwhile, Dan's session is still active, his session state object is included in shared memory.

To allow for quick responsiveness to user requests, the application stores some key information in shared memory. Figure 3-3 diagrams the different shared memory object, their relations and what information they store.

A *Mint Server State* object that has null labels is stored as the Aeolus *root* object and contains the following objects:

Bank Closures Map

A bank closure is used to retrieve user information from a bank. The bank closure is authoritative for the tag of the bank it is working for. A bank closure has null labels. This object maps each bank name to a bank closure, and has null labels itself.

Session Objects Map

This object maps a username to a session object. A *session object* contains session information about a user. It also contains a list of *bank credential* objects, one for each bank a user has registered for, as well as the user's Principal ID (PID). A *session object* has the *USER-DATA* tag in both its labels.

Bank Credentials

A *bank credentials* object contains the user's username and password for that particular bank. It has a *BANK-DATA* tag and a *USER-DATA* tag in its secrecy label, and a *USER-DATA* tag in its integrity label.

User Tags Map

The application stores the tag for each user that has an active session, this allows the application server to access session objects with the user tag in their label. User tags are stored as a mapping from username to tag. This object has null labels.

Bank Tags Map

Bank tags are important to store in memory to allow for quick server response time. They are stored as a map from bank name to bank tag. This object has null labels.

User Last Activity Map

The time of a user's last activity is used to identify which users are still active. This information is stored as a mapping from username to last activity time. This object has null labels.

3.2 Implementation

The application server runs on a VN, and receives user requests via RPC. Users can sign up, login, logout, register a bank and retrieve expenditure statistics.

At startup, the system registers a number of banks. These are the banks which

the users can later add accounts for on their Mint account. The system is now ready to accept users.

Users can register to the Mint service through a *signUp* RPC call:

signUp(username, password)

Signs up a new user with the username as *username* and password as *password*. If *username* is available, associates those credentials with a newly-created user data tag and a newly-created principal ID, then stores all this information on disk under a new user directory, as per the file system hierarchy described in section 3.1.2. The system also stores the newly created tag in the *User Tags Map*. Throws an exception if this *username* is already used.

Users can then log in to modify their account and retrieve their information using the following RPC call:

login(username, password)

Authenticates a user, and if successful, stores their information in a session object, and updates their last activity time using the shared memory objects described in section 3.1.3. The system authenticates the credentials by calling an *authentication closure*, which adds the All User Data tag, verifies the credentials, declassifies and terminates if the *username* and *password* are correct, throws an exception otherwise.

Once a user logs in, they only have to include their *username* in pursuing requests to carry out different actions. In each of the following RPC calls, authentication is carried out by checking if the provided *username* has a session that has not timed out yet (there is a last activity time associated with a session, and thus must not be older than a certain amount of time, e.g. 15 minutes).

addBank(username, bankName, bankUsername, bankPassword)

Registers the specified bank for the specified username with the given credentials. This request writes a *user-bank-creds* file to disk, with the tag of the user

in its integrity label, and the tag of the user and that of the bank in its secrecy label.

downloadTransactions(username)

This action connects to each bank the user has registered for their account, downloads the latest transactions for the user, processes them, and returns the result. The RPC thread uses the closure of the bank to connect to the bank and declassify the information returned. The thread then adds a user tag to its secrecy label, and uses a reduced authority call to the public principal to process the information; this ensures that information cannot be leaked while it is being processed [2].

logout(username)

Terminates the user's session, removing their information from shared memory.

removeBank(username, bankName)

Removes the specified bank from a user's account.

For each RPC call, the thread handling the call starts running on behalf of P_{MINT}, and then reduces its authority as soon as possible to avoid errors leading to information leakage. For example, in *downloadTracations*, after authenticating the user, the thread reduces its authority to run on behalf of the users principal, then processes the downloaded information on behalf of P_{PUBLIC} and then returns.

3.3 System Security

The use of a *username* in the implementation of the RPC calls defined above to identify which user invoked those calls creates a vulnerability in our system. Alice could send a *downloadTransactions("Bob")* request, and because the information is being sent outside the Aeolus system, the information in the reply would have null labels, and hence Alice could access Bob's information.

[2] One can imagine that the information is being processed by a different application in the system, using this technique means our application does not have to trust that application.

Another problem that comes up is related due to the assumption that the user is connecting from another Aeolus node. If this was not the case, i.e. the requests are coming from outside the Aeolus system, sensitive information would be exposed over the network (e.g. the user's Mint credentials), which wouldn't happen for RPCs within the Aeolus system because those are encrypted[3].

Both issues can be resolved by using standard cryptography mechanisms to encrypt and authenticate external communication. For example, encrypting all requests and replies coming in and out of the system and, in addition, sessions can be identified by unguessable temporary session-ids, and, those session-ids would be used to authenticate future requests in that session.

Finally, the security of application can be improved by introducing more principals to the authority state model to make the information flow control more fine-grained. For example, we could separate the authority to register banks from the authority to authenticate users.

[3]Requests coming from outside the system could be RPC or HTTP requests.

Chapter 4

Auditing the Mint Application

In this chapter we discuss auditing and how the audit trail can be used for discovery.

As described in section 2.3.5, every security-related event is logged automatically by Aeolus. In addition, applications can log events of their own. We take advantage of this in the Mint system by logging application-specific events; an example is given in section 4.1.

Events are stored in a table that we will refer to as the *LOG*. The LOG is useful for discovery, which can be done by means of a query over the table. Table 4.1 and figure 4-1 show two examples.

secrecy	integrity	user	op_name	timestamp
{11}	{}	Alex	DECLASSIFY	2012-06-13 21:57:06.876
{11}	{}	Aline	DECLASSIFY	2012-06-13 21:57:04.852
{11}	{}	Jack	DECLASSIFY	2012-06-13 21:56:45.73
{11}	{}	Jack	DECLASSIFY	2012-06-13 21:56:42.691
{11}	{}	Mike	DECLASSIFY	2012-06-13 21:56:40.649
{11}	{}	Mark	DECLASSIFY	2012-06-13 21:56:39.624

Table 4.1: Bob's Information Leaks. This table shows the declassifies of Bob's data tag from users other than Bob himself. The columns are as described in section 2.3.5.

Table 4.1 shows principals that declassified Bob's tag; thus, it shows which users accessed Bob's information. In a secure implementation of the Mint application, no user principal other than Bob's principal should be able to declassify Bob's tag. How-

ever we introduced a bug in our implementation of the Mint application to demonstrate how the LOG can be used for discovery, and hence we can see that many user principals were able to declassify Bob's tag - something that should never happen. In this example, the LOG helps us discover the leakage of Bob's information due to an application error.

Figure 4-1 shows a graph of how many user requests, for each user, caused a declassification of a user data tag for every 5-second period since the application launched; in other words, it shows the activity trend of each user. Any spike in the graph for a particular user means that that user was abnormaly active in certain periods of time. In this example, the LOG helps us discover if any user accounts have been compromised.

In the following section we describe how the data for Figure 4-1 can be extracted from the LOG.

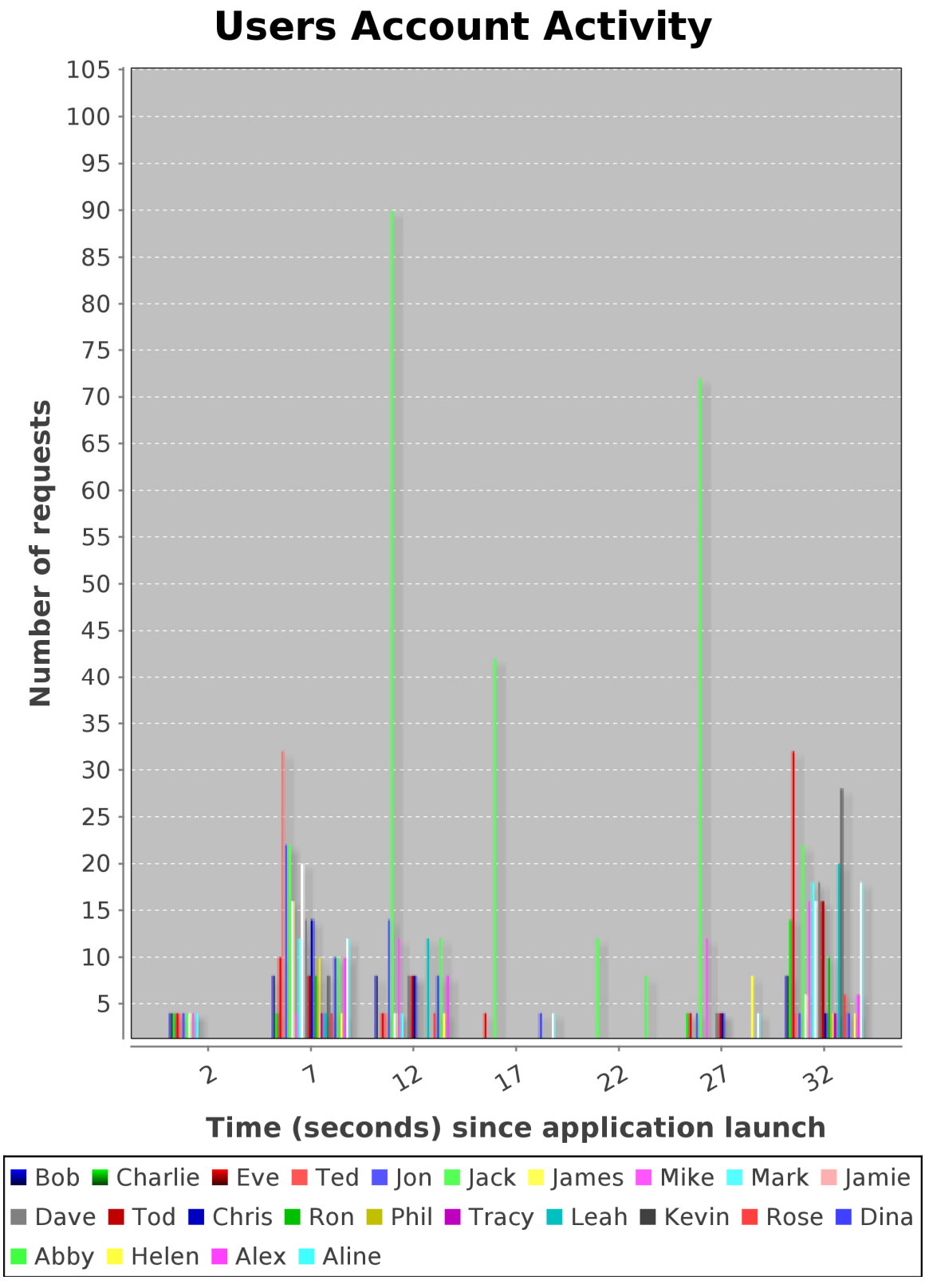

Figure 4-1: This graph shows the activity trend of each user. Notice that the activity for Jack spikes in the graph at multiple places. This could be the result of an attacker accessing his account and issuing requests to quickly retrieve all of Jack's financial information.

4.1 Detecting Suspicious User Activity

In order to gather the information necessary to produce the graph in figure 4-1, we add application-specific events to the LOG for each different action the user takes.

For example, when Bob signs up successfully with our application, we record his username, "Bob", as well as his data tag and the principal authoritative for that tag. This is done using the following call[1]:

```
AeolusLib.createEvent(``mint-user-signup'',
  Arrays.asList(
    new String[] {``Bob'', ``5'', ``10''}));
```

This call causes a *sign up* event to be added to the LOG. To obtain a table detailing the username, principal, data tag and other sign-up information of all users, we use the following SQL query based on our application-specific events[2]:

```
select to_array(app_arg)[0] as pid,
  to_array(app_arg)[1] as username,
  to_array(app_arg)[2] as tag,
  timestamp as signed_up,
  event_counter
from events
where op_name='mint-user-signup'
```

Table 4.2 shows a possible result of running the query, with some extra information about the time when each user signed up.

Finally, we can obtain the information necessary to create the graph in figure 4-1 using the following query:

```
select e.count, users.username,
  e.to_timestamp as timestamp
from (select count(*), tags_modified,
```

[1]Recording the principal ID and the number of the tag is enough to retrieve the full information about a principal and a tag from the AS.

[2]*to_array* is a procedure that turns a comma-delimited string into a PSQL array.

pid	username	substring	signed_up	e_ctr
6	wissam	4	2012-06-13 21:56:36.207	131
9	david	7	2012-06-13 21:56:36.207	445
12	dan	10	2012-06-13 21:56:37.338	793
13	Bob	11	2012-06-13 21:56:37.338	1148
14	Charlie	12	2012-06-13 21:56:37.338	1211
15	Eve	13	2012-06-13 21:56:37.338	1274
16	Ted	14	2012-06-13 21:56:37.338	1337
17	Jon	15	2012-06-13 21:56:37.338	1400
18	Jack	16	2012-06-13 21:56:37.338	1463
19	James	17	2012-06-13 21:56:37.338	1526
20	Mike	18	2012-06-13 21:56:37.338	1589
21	Mark	19	2012-06-13 21:56:37.338	1652
22	Jamie	20	2012-06-13 21:56:38.578	1715
23	Dave	21	2012-06-13 21:56:38.578	1778
24	Tod	22	2012-06-13 21:56:38.578	1841
25	Chris	23	2012-06-13 21:56:38.578	1904
26	Ron	24	2012-06-13 21:56:38.578	1967
27	Phil	25	2012-06-13 21:56:38.578	2030
28	Tracy	26	2012-06-13 21:56:38.578	2093
29	Leah	27	2012-06-13 21:56:38.578	2156
30	Kevin	28	2012-06-13 21:56:38.578	2219
31	Rose	29	2012-06-13 21:56:38.578	2282
32	Dina	30	2012-06-13 21:56:38.578	2345
33	Abby	31	2012-06-13 21:56:38.578	2408
34	Helen	32	2012-06-13 21:56:38.578	2471
35	Alex	33	2012-06-13 21:56:38.578	2534
36	Aline	34	2012-06-13 21:56:38.578	2597

Table 4.2: Mint Users Information. This is a table detailing the information about users signing up during a simulated run of the Mint application. The *pid* column denotes the user's principal, the *username* column denotes the user's username, the *tag* column denotes the user's data tag, the *signed_up* column denotes the time at which the sign up event took place, and the *e_ctr* column denotes the unique event_-counter number of the sign up event.

count	username	timestamp
22	Jack	2012-06-13 21:56:40-04
4	Jack	2012-06-13 21:56:35-04
42	Jack	2012-06-13 21:56:50-04
72	Jack	2012-06-13 21:57:00-04
90	Jack	2012-06-13 21:56:45-04
12	Jack	2012-06-13 21:56:55-04
22	Jack	2012-06-13 21:57:05-04

Table 4.3: Mint Account activity for user Jack. This table shows the number of declassifies done by Jack's principal in each five-second period since the application launched. The *count* denotes such number. The *username* column denotes the user who's data tag is in question. The *timestamp* column denotes the start of the 5-second period. A table containing similar information for all users would be a precursor to producing the graph in figure 4-1, but here we only show this data for user Jack for simplicity.

```
(to_timestamp(((extract(epoch
  from events.timestamp)/5)::int)*5))
from events where tags_modified
in (select tag from users) group by
(to_timestamp(((extract (epoch
  from events.timestamp)/5)::int)*5)),
tags_modified)
as e inner join users on e.tags_modified=users.tag
```

Table 4.3 shows a the result of a modified version of this query that shows information just for the user Jack. This query provides us with a count of how many requests by Jack caused a declassification of a user data tag for every 5-second period of time since the application launch.

Chapter 5

The Summary System

This thesis is concerned with providing mechanisms that allow the size of the log to be controlled. A log for a large system (e.g., the Mint application with millions of users) is going to be very big, and the problem gets worse if the system is long-lived. A way to reduce the log is to delete or archive events. Typically, this will be done for events in the past, e.g., events collected last year are less important than recent events. However, there is no way that Aeolus can decide by itself what events are important: this has to be done by the application. A further point is that the application may need to retain some of the information in the events, at least those that are being deleted; again the application has to specify what information should be retained.

For example, if we store the number of declassifies a user's principal carries out per user session, we would require many fewer events to be stored in the log table to formulate a pattern of the user's activity, and, eventually detect suspicious behavior patterns. Thus, we could choose to delete events that have been summarized, without loss of the summarized information.

Our system is designed to support this process. We provide an interface that can be used by application code to create *summaries* (also referred to as *summary events*) and to mark events for deletion or archiving; from this point forward we refer to these as the *marked* events. Summaries are stored in a *SUMMARY* table and therefore queries can be used to perform discovery on them if desired.

5.1 Summarization Model

An application uses our summary system from within a VN. This has the following ramifications:

- Every access to the LOG or SUMMARY table must obey our information flow constraints. The thread can only read events from these tables whose secrecy labels are contained in those of the thread, and vice versa for integrity labels. Also, all events that are added to the SUMMARY table will have the labels of the thread at the moment the event is added.

  The thread producing the summaries or deletes runs with the authority of some principal. For example, a thread creating a summary in the Mint application would likely run the P_{MINT} principal. Thus the thread can use its authority to declassify as needed to produce a summary with the appropriate labels.

- The user code can run an arbitrary Java computation to produce the summary from the events being summarized. This computation could be a simple one, such as a count of the number of the DECLASSIFY events encapsulated by a summary, or a more complex one, such as a categorization of the users activity as normal or suspicious based on the different events encapsulated by the summary.

- The activities of the user code will be audited in the usual way. Thus it will be possible to run discovery over these events, and if desired the user code can log additional events in the base log by using the Aeolus interface.

In the following section, we describe the typical use cases that our system aims to address.

5.2 Summarization Workflow

A typical workflow of how an application might use our system to produce a summary is as follows:

1. Possibly run some queries to establish a context for the events to be summarized or marked. For example, identifying Jack's principal ID is necessary to summarize the number of declassifications carried out by Jack's principal in a particular session. Such a context could be established by retrieving a table detailing each user's information as shown in table 4.2[1].

2. Run a query to identify the events to be summarized or marked, for example, all the declassifications carried out by Jack's principal per session.

3. Compute the information that goes into the summary, for example, whether or not the number of declassifications caused by Jack's principal for a particular session is too high.

4. Possibly produce and store the summary.

5. Possibly mark all events covered by the summary.

The next two sections describe how our system allows applications to query, summarize and mark events.

5.3 The Query System

Steps 1 and 2 above are used to establish the context and then to identify the events to be summarized or marked. Since these activities are likely to be performed periodically, it is useful to allow the queries to be defined once and then used over and over.

For example, the queries in section 4.1 could be run many times over the lifetime of the application.

However, an application might need to use a particular query with different parameters - for example, the Mint application might want to run a query to find out the last login time for different users.

[1]Other necessary information for this summary example is to identify the start and end of Jack's sessions.

A further point is that queries could be run over the LOG or the SUMMARY table, or both. It is up to the application to specifically identify which tables the query will run over.

The following two sections describe the interface we provide to support these use cases, as well as the *QUERY* table which we use to store query records.

5.3.1 *QuerySystem*

An application can use our system by calling QuerySystem.getInstance(). This will provide the application with an instance of our system, which can then be used to call the following methods for managing and running queries in the QUERY table:

addQuery(name, query_text)

Adds a query with a NAME as *name* and TEXT as *query_text*. If a query with that name already exists, throws an exception.

runQuery(name, vals)

We allow applications to provide incomplete query text (values replaced by placeholders) to allow for reuse of the queries with different parameters.

This method runs the query with the name *name*, replacing any existing parameter placeholders in the query TEXT with the values in the *vals* array, and returns the underlying set of event records.

The *addQuery* and *runQuery* methods log the type of the operation, e.g. "query-add" or "query-run", as well as the name (and query parameters in the case of *runQuery*) of the query involved.

We do not support deletion of queries from the QUERY table because summaries could refer to them, as described in section 5.4.

A final point is that *runQuery* will return only records that the calling thread is allowed to see per the rules in section 2.2.2.

5.3.2 Query Attributes

For each record stored in the QUERY table, we store the following attributes:

NAME

This field is provided by the application. It can be used to identify the purpose of the query, for example, query 4.1 could be named "Mint Sign-ups".

TEXT

This is the text of the query. This does not have to be a complete query: again, we allow the application to leave query parameters unspecified, replaced with placeholders in this field, so that a record can be re-used with different parameters to represent different queries. The following is an example of what this attribute might hold as a value:

```
select * from LOG where event_counter=?
```

We do not store labels in the QUERY table because we expect those queries to run over the LOG and SUMMARY tables, which already have labels. To avoid any confusion and prevent the use of our interface as a covert channel, we require that a thread have null labels when adding a query to the QUERY table.

5.4 Producing Summaries and Marking Events

An application can use queries in the QUERY table to set up a context for summarization and to identify events to be summarized. The application can then store a summary of those events in the SUMMARY table. The application might then choose to mark the events encapsulated by a particular summary event, or summarize a group of summary events.

For example, a summary of the information presented in figure 4.1 would detail all the users other than Bob who accessed Bob's information, along with the period of time during which those accesses happened (identified by a start event_counter and an end event_counter).

The following two sections describe the interface we provide to support producing summaries and marking events, as well as the SUMMARY table which we use to store summary records.

5.4.1 *SummaryObjects*

In this section we describe *SummaryObjects*. *SummaryObjects* provide an interface for applications to create and add summaries to the SUMMARY table, as well as mark their events.

The interface we expose is as follows:

SummaryObject(query_name, query_params)

Retrieves the events identified by the query *query_name* with parameters *query_params*.

addSummary(summary_name, summary_args)

Adds the summary represented by this *SummaryObject* to the SUMMARY table with name as *summary_name* and arguments as *summary_args*. Stores the calling threads' secrecy and integrity labels of the summary object.

getEvents()

Returns a set of records that this summary object covers. This set of records is determined when the summary object is created.

mark(query_name, query_params)

Marks the group of events underlying this *SummaryObject*. The *query_name* and *query_params* identify a subset of the events covered by this summary object that should NOT be marked. If the arguments are *null*, all events covered by this summary object will be marked.

The reason we allow some events to be identified as not markable is because there are events in the log that Aeolus won't allow to be deleted; in particular all events that record changes to the authority state, e.g., creating a principal, or allowing one principal to act-for another. It seems likely that the application will also have some

events that it does not allow to be deleted. The arguments to the mark method allow these events to be identified.

In our scheme, we depend on being able to identify events underlying a particular summary by their event_counter. Hence we require all queries used to produce *SummaryObjects* to include an *event_counter* attribute.

Finally, The *addSummary* and *mark* methods log the type of the operation, e.g. "summary-add" or "summary-mark", as well as the name and parameters of the underlying query.

5.4.2 Summary Attributes

For each record stored in the SUMMARY table, we store the following attributes:

EVENT_COUNTER

Unique identifier for each record.

NAME

Similar to *NAME* in the QUERY table, this field is provided by the application and could be used to identify the content of the summary record.

ARGS

This field is provided by the application. It stores the application's computation of the summary. In the example mentioned above, this could be the list of users accessing Bob's information over a particular period of time.

QUERY_NAME

This field is provided by the application. It identifies the query in the QUERY table that identifies the events summarized by this record.

QUERY_PARAMS

This field is provided by the application. It stores the parameter values for the query identified by the *QUERY_ID* if any.

SECRECY

This is the secrecy label of the thread that added the summary record.

INTEGRITY

This is the integrity label of the thread that added the summary record.

TIMESTAMP

The real time at which this summary was created.

5.5 Marking Events

Our system allows but does not require the production of a summary event in conjunction with marking. For instance, a *SummaryObject* can be created, then used to mark events, without the application having to call *addSummary*.

Since our system was not intended to alter the LOG or SUMMARY table, we keep track of marked events in two additional tables: LOG-MARKED and SUMMARY-MARKED. Those tables contain event counters of events that have been marked by the application.

Our system does not show marked events in the results of later queries. Since those events are not deleted, and could be made visible to the user, although our system does not do this at present.

Finally, our system only provides a mechanism to identify marked events, and leaves the question of how to archive and delete those events as further work.

5.6 Discussion

It is possible that an application uses our system that to summarize pre-existing summaries. For example, the Mint application, after summarizing declassify events for user data tags per user session, could use those summaries to summarize declassifiy events for user data tags per month.

Our system supports this functionality since it does not make any strong assumption about queries in the QUERY table. This allows the application to treat events in the LOG or SUMMARY table the same way while summarizing.

Finally, by allowing applications to protect events from being marked, and internally preventing important Aeolus events from being marked, our system provides a way to prevent application errors from deleting important information.

Chapter 6

Auditing Using Summarization

In this chapter we give examples of how our system can be used to summarize and access information for the Mint application, as well as mark certain events.

6.1 Summarizing User Sessions

In this example we show how to summarize a user's actions per session. In particular, for a session for the user Bob, we will record which of the user actions presented in section 3.2 (e.g. addBank, downloadTransactions) he requested.

This should not be a hard task since we use application-level logging to track all of Bob's actions. However, we will need to create some context before producing the summaries to identify any one of Bob's sessions.

First, we initialize the query system, which we will use to add and run queries:

```
//QuerySystem uses the singleton paradigm
QuerySystem system = QuerySystem.getInstance();
```

We then set up a query that retrieves the princpal ID and data tag for a particular user. Note that in our implementation, the name of the LOG table in the database is *events*.

```
String users_query = ``select
    to_array(app_args)[1] as pid,
```

```
    to_array(app_args)[2] as tag,
    event_counter
  from events
  where app_op='mint-user-signup'
    and to_array(app_args)[0] = ?'';

system.addQuery(``user-info'', users_query);
```

This query is based on logging events when users sign up as described in section 4.1. The *app_args* attribute contains three pieces of information; the username, principal ID, and data tag, as a comma-delimited string. The query produces a tuple with the principal ID in the first column, data tag in the second column and event_counter of the signup event in the third column for a particular user. It does so by looking for signup events, which have the attribute *app_op* as "mint-user-signup", for a particular username, stored in the first index of the *app_args* attribute.

We now add some queries that will help identify the logins and logouts for a particular user:

```
// When a user, say Bob,  logs in, we execute the
// following code:
/*
AeolusLib.createEvent(
 ``mint-user-login'', Arrays.asList(
 new String[]{``Bob''}));
*/
// and similarly for logouts.

// The following query produces a tuple that contains
// the login
```

```
// times and event_counters for a particular user.
String user_logins = ``select
    timestamp as start,
    event_counter
  from events
  where app_op='mint-user-login'
    and app_arg[0]=?'';

// The following query produces a tuple that
// contains the logout
// times and event_counters for a particular user.
String user_logouts = ``select
    timestamp as start,
    event_counter
  from events
  where app_op='mint-user-logout'
    and to_array(app_args)[0]=?'';

// add the queries to the QUERY table
system.addQuery(``logins'', user_logins);
system.addQuery(``logouts'', user_logouts);
```

Users cannot login if they are already logged in, and they are logged out after a certain period of time. Hence, for events far enough in the past, we can assume that there is a one-to-one pairing between user login events and user logout events. We can now use the information from those queries to identify Bob's sessions:

```
// A ResultSet is a set of records.
ResultSet bob_logins =
  system.runQuery(``logins'', new Object[]{``Bob''});
ResultSet bob_logouts =
```

```
  system.runQuery(``logouts'', new Object[]{``Bob''});

// The first login and logout for Bob will
// define the endpoints of the first session.

if((!bob_logins.next()) || (!bob_logouts.next())){
  return;
}

// Information for the first session
String username = ``Bob'';
long start_counter = bob_logins.getLong(3);
long end_counter = bob_logouts.getLong(3);
Timestamp start_time = bob_logins.getTimestamp(2);
Timestamp end_time = bob_logouts.getTimestamp(2);
```

A ResultSet is a set of records. Calling the *next()* method on the ResultSet objects move their cursor to the first record. We first make sure that Bob logged in to our system once in the past and then logged out. We then retrieve information from the ResultSet using the getLong() and getTimestamp(). These methods retrieve a value at the given column number from the ResultSet and cast those values to their relevant types. Details of the ResultSet class and how its methods work can be found in [11].

Table 6.1 shows a possible table that contains a list of user sessions. We are now left with the problem of identifying the different actions that took place within any sessoin (identifiable by a start event_counter and an end event_counter).

```
String user_session_actions_query = ``select
    app_op
  from events
  where app_op like 'mint-user-\%'
    and to_array(app_args)[0] = ?
```

username	start_counter	end_counter
wissam	168	373
david	482	1075
dan	830	1086
Ted	2644	22675
Leah	2690	19437
Alex	2736	7786
Jamie	2782	5991
Dina	2828	7336
Mark	2874	18576
Helen	2920	6312
Jon	2966	18370
Mike	3072	18321
Jack	3118	15342
Eve	3164	7735
Charlie	3210	7434
Phil	3450	19535
Tod	3684	5895
Chris	3924	18075
James	3970	7635
Dave	4110	18223
Aline	4253	18773
Rose	4580	17816
Ron	4626	19699
Kevin	4891	17865
Tod	5931	17714

Table 6.1: User Sessions. This table shows start and end event counters for different user sessions. Start and end times are omitted for simplicity.

```
    and event_counter > ?
    and event_counter < ?'';

system.addQuery(``user-sessions-actions'',
user_sessions_actions_query);
```

Now that we've identified one of Bob's sessions, i.e. the first one, and have a query to find out his actions during that session, we can go ahead and summarize Bob's actions during his first session. For each session we produce a summary that contains the username, the actions carried out, and the event_counters and timestamps that identify the start and end of that session.

In this example, and the ones that follow, we assume that the method *process(events)* is a user-defined method that processes a set of events and produces a string summary of those events. In this example, such a summary string would be the comma-delimited list of activites requested by the user in this session. For example, if a user registers Bank of America to their account and downloads all of their transactions in one session, the *process()* method will produce the following summary for that session "addBank-bankofamerica,downloadTransactions".

```
Object[] vals = new Object[] {username,
  start_counter, end_counter,
  start_time, end_time};
SummaryObject so = new SummaryObject(
  ``user-session-actions'', vals);

String summary = process(so.getEvents());
so.addSummary(``user-session-summary'',
String.format(``\%s, \%d, \%d, \%s, \%s'',
  username, start_counter, end_counter, start_time,
  end_time, summary));
```

In this section, we showed how to produce a summary for Bob's actions for his first session. This summary would contain Bob's username, start and end timestamps and event counters of his first session, as well as all Bob's actions in that session.

6.2 Summarizing File System Events

In this example we explore how to summarize information about file system events; we show how to store information about file accesses in the last week.

```
String fs_access = ``select *
  from events
  where filename<>''
    and roundweekdown(timestamp)'';

system.addQuery(``fs_week_access'', fs_access);

long now = System.currentTimeMillis();
Object[] vals = Object[]{new java.sql.Timestamp(now)};
SummaryObject so = new SummaryObject(``fs_week_access'',
vals);

String summary = process(so.getEvents());
so.addSummary(``File accesses'',
String.format(``\%d, \%s'', now, summary));
```

The above code will store a summary of the file system events in the last week[1]. Only file system events use the *filename* attribute in the LOG, hence we use that to retrieve all file system events[2].

[1] *roundweekdown* is a function that rounds down the timestamp to the start of the last week. It does the same as the query in section 4.1 which groups timestamps to 5-second periods (except this one is a one week period).

[2] The *DATE_SUB* substracts an interval of time from a date, as described in [13].

After producing the summary, we might wish to mark most file system events, except for the ones that change the file system hierarchy (adding or removing directories or files). We could do this as follows:

```
String fs_protect = ``select * from events
where op_name = ? or op_name = ?''
system.addQuery(``protect-fs-events'', fs_protect);
so.mark(``protect-fs-events'',
  new Object[]{``CREATE_DIRECTORY'',
    ``REMOVE_DIRECTORY'', ``CREATE_FILE, ``REMOVE_FILE''});
```

6.3 Summarizing User Trends

As a final example, we will show how records in the SUMMARY table itself could be summarized, producing *summaries of summaries*. In this section we will further summarize session summaries. To build on what we encountered in section 6.1, let's assume that more of Bob's sessions have been summarized. We can now coalesce them to detail Bob's activity per week. The following code shows how to do so for the last week.

```
String session_summary_query = ``select *
  from SUMMARY
  where name=user-session-summary
    and to_array(args)[0]=?
    and to_array(args)[3]::timestamp > DATE_SUB(?,
      INTERVAL 7 DAY)
    and to_array(args)[3]::timestamp < DATE_SUB(?,
      INTERVAL 14 DAY)'';

system.addQuery(``session-summary'', session_summary_query);
```

```
Timestamp now = new Timestamp(System.currentTimeMillis());
Object[] vals = new Object[]{``Bob'', now};
SummaryObject so = new SummaryObject(``session-summary'',
  vals);

// Create a summary based on the sessions in the last week
String summary = process(so.getEvents());
so.addSummary(``weekly-user-activity'', summary);
```

6.4 Sufficiently Summarized Information

The summaries produced in sections 6.1 and 6.2 overlap in what they summarize. For example, adding a *user-bank-creds* file to the file system happens only when a user adds a bank to their account, i.e. when they request the *addBank(...)* user action. Hence this particular piece of information, that a user 'A' registered bank 'B' to their account, could be concluded from two summaries: one that summarizes the user's actions per session (or per week), and one summarizing the file system operations.

There are many other examples in which summaries could overlap in what they summarize, and it is one of the reasons why we choose to leave it up to the application to decide if and when certain events are "sufficiently summarized", and are ready for marking, rather than trying to determine that ourselves.

Chapter 7

Implementation

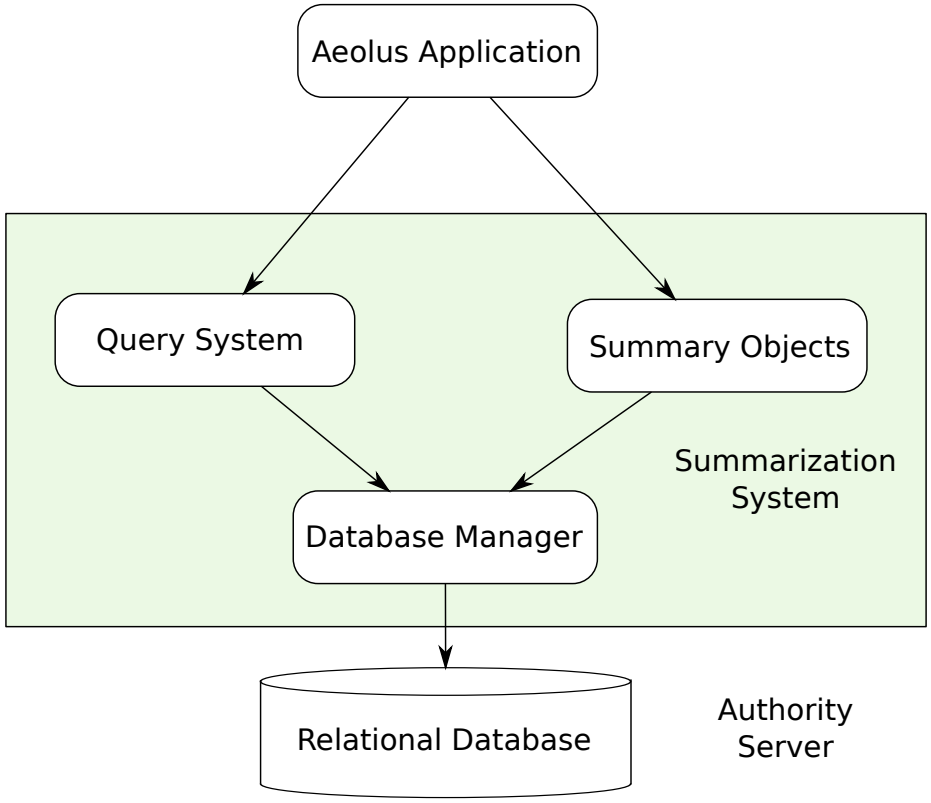

Figure 7-1: Summarization System Architecture. This diagram illustrates the components involved in the summarization system we provide.

The codebase for this thesis is implemented in approximately 4000 lines of Java code, of which 3000 implement the Mint application and 1000 implement the summarization system. The focus of this chapter is the implementation of the summarization

system.

Figure 7-1 presents a high-level overview of the summarization system components. The application uses the Query System (QS) and Summary Objects (SOs) to produce summaries and mark events. The Database Manager (DBM) stores queries and summaries in a PostgreSQL relational database [13], as well as retrieves results of queries. On VN startup, we create the extra QUERY, SUMMARY, LOG-MARKED and SUMMARY-MARKED tables necessary for the running of the summarization system.

The rest of this chapter describes the implementation of those components.

7.1 The Query System and The Summary Objects

The QS provides the interface described in section 5.3.1 and a Summary Object provides the interface described in section 5.4.1. Summary Objects store the name and parameters of the underlying query locally.

Both the QS and the SOs are responsible for logging calls to their methods as described in sections 5.3.1 and 5.4.1.

7.2 The Database Manager

All communications to the relational database go through the DBM. The DBM provides an internal interface for the QS and SOs to carry out the various operations necessary.

setUpSummarizationSystem()

This method creates the

This method is called by the QS.

addQuery(name, query_text)

Executes a SQL INSERT statement to add a record to the QUERY table with

the values *name* and *query_text*. Throws an exception if the labels of the calling thread are not empty.

This method is called by the QS.

runQuery(name, vals)

This call carries out the following actions, in order:

1. Executes a SQL SELECT statement to retrieve the query text from the QUERY table.
2. Uses a Java PreparedStatement to combine the query text with the values in the *vals* array, and throws an exception if the expected number or type of parameters do not match the ones provided.
3. Modifies the query to exclude any marked events. For example, if the combined query text was

   ```
   select * from events where
   app_opp='mint-user-signup'
   ```

   the DBM modifies it to the following

   ```
   select * from (select *
   from events where app_op='mint-user-signup')
   as e where e.event_counter not in
   (select event_counter from events-marked)
   ```

   We use a subquery to carry out the task to avoid having to parse the query for a an already-existing where clause in the query.
4. Executes the combined query text and retrieves the set of records covered by that query.
5. Removes from the retrieved set of records any records that the calling thread is not allowed to view per the DIFC rules described in section 2.2.2, then returns the resulting set of records.

This method is called by the QS and the SO constructor.

addSummary(s_name, s_args, q_name, q_ args, tstamp)

s_name is the summary name, *q_name* is the query name (similarly for *s_args* and *q_args*. *tstamp* is the timestamp. Executes a SQL INSERT statement to add a record to the SUMMARY table, using the arguments and labels of the calling thread as as values for the record.

This method is called by the SOs.

mark(query_name, query_params)

Executes a SQL INSERT statement to add events to the LOG-MARKED and SUMMARY-MARKED tables to identify marked records. Event counters in the SUMMARY table contain a non-numeric character to distinguish them from event counters in the LOG. The DBM uses this knowledge to distinguish where to add a marked event (LOG-MARKED or SUMMARY-MARKED).

This method is called by the SOs.

In an ideal implementation, our system would be using the PostgreSQL Information Flow Control database provided by David Schultz [14]. This would push the tasks of enforcing empty labels when adding queries and filtering query results according to the label of the calling thread to the database itself to handle.

7.2.1 Running in *Hide* Mode

Once events are marked, they are hidden from the application in future queries. This is implemented by introducing a two views, LOG-UNMARKED and SUMMARY-UNMARKED, that only contains events that are in the LOG and SUMMARY tables but not in the LOG-MARKED and SUMMARY-MARKED tables. We expose only the names of the LOG-UNMARKED and SUMMARY-UNMARKED views to the application for writing queries.

Our system can be extended to implement a *no-hide* mode, where the LOG-UNMARKED and SUMMARY-UNMARKED views show exactly the same records as the LOG and the SUMMARY tables.

Chapter 8

Related Work

Dealing with large audit trails is a problem that security platforms other than Aeolus face as well. In our summarization system, we allow the application to summarize events in the audit trail before deleting or archiving them.

The intrusion detection system BackTracker [8] allows applications to track intrusion points into a system by analyzing operating system events, which are represented as a dependency graph. BackTracker also provides an analysis tool, GraphGen, to analyze those dependency graphs. As those dependency graphs get larger, GraphGen allows a system administrator to trim certain pre-defined events from analysis, as well as allows the administrator to define further events to be trimmed by use of regular expressions. Whereas trimming events allows for dependency graphs to be analyzed faster, there is an associated risk of losing important information when events are trimmed. Furthermore, those events are never fully removed from the system, they are only removed from the analysis tool, and hence the storage space for the audit trail isn't reduced. In contrast, our summarization scheme makes it safer for applications to remove events after summarizing them, while still allowing for the important information contained in those events to be available for analysis through the event summaries.

The importance of allowing systems that rely on audit trails to reduce the size of the audit trail while still maintaining important information is well-recognized in the research community ([6, 2, 9]). However, not many systems have been built to address

this problem ([1]). Possible uses of data compression [1] and artificial intelligence [7] techniques have been surveyed and discussed to in the past, but to our knowledge none has been implemented.

Chapter 9

Summary of Contributions and Future Work

9.1 Contributions

In this thesis, we present a summarization system that allows applications to summarize information in their Aeolus audit trails, as well as mark events for later archiving or deletion. We also present a query system that makes it easier for applications to both use our summarization system and query the Aeolus audit trails in general.

Note that summarization presents information in a condensed form. If that information is what is needed to run discovery queries, running these queries over the SUMMARY table will provide better performance than having to fetch all the underlying events from the LOG.

9.2 Future Work

The are two main directions in which our system could be extended in the future.

The first is improving the expressive power and performance of the current system. An example of the former is that it might be useful to define a "no-mark" query just once and have it apply to every marking to rule out certain events from being marked. Note that this matches how we do things in Aeolus (where events recording

modifications of the authority state are never marked). This approach could replace the current arguments to the *mark* method described in section 5.4.1, or it could be used in conjuction with those arguments; the latter approach would allow the application to identify events to not mark in a way that is specific to a particular call of the *mark* method.

An example of the possible performance improvement is to provide the ability for the application to have materialized views. For example, in the Mint application, it would be really useful if the application could define a view that would produce the signup information of all users (shown in table 4.2) whenever requested. The presence of such a view could greatly speed up later uses of that query.

The second direction to extend our system is to allow applications to handle archiving and deletion. Right now we only mark events but we do not archive or delete them. Ultimately, though, it will be necessary do support those functions. But again the application will have to be specific about what to do: some marked records might be deleted, some might be archived, and some might be archived and then deleted. A possible direction to providing archiving and deletion is to allow the application to define particular queries to do the delete or archive, e.g., there could be a query that deletes all base events for sessions for users who no longer use mint and where those sessions are older than a year, or a query that deletes all but the events that add and remove banks, and archives those events.

Note that if events are archived we might also like to allow them to be fetched. The idea is that the user might want to "drill down": they have obtained information from the summaries and have decided they need to see the specific events that led to that summary. This is another feature that such an arhiving and deletion system should include.

References

[1] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, and Ed Stoner. State of the practice of intrusion detection technologies. Networked Systems Survivability Program CMU/SEI-99-TR-028, Carnegie Mellon University, CMU Pittsburgh, PA 15213-3890, January 2000.

[2] Edward Roback Barbara Gutman. *An Introduction to Computer Security: The NIST Handbook*. National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, oct 1995.

[3] Aaron Blankstein. Analyzing audit trails in the Aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, June 2011.

[4] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012.

[5] 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.

[6] Gene Spafford et al. Audit trail reduction. Online, `http://www.cs.purdue.edu/coast/projects/audit-trails-reduce.html`, 1999.

[7] Jeremy Frank. Artificial intelligence and intrusion detection: Current and future directions. June 1994.

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

[9] Teresa F. Lunt. A survey of intrusion detection techniques. *Computers and Security*, 12(4):405 – 418, 1993.

[10] F. Peter McKee. A file system design for the aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, September 2011.

[11] Oracle. *Java Platform, Standard Edition 7 API Specification*, 2012.

[12] 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.

[13] PostgreSQL Development Team. *PostgreSQL 9.0.4*. PostgreSQL Global Development Group, 2010.

[14] David Schultz. *Decentralized Information Flow Control for Databases*. Ph.D., MIT, July 2012.