Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-040 August 27, 2009

Information Flow for Secure Distributed Applications

Winnie Wing-Yee Cheng

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Information Flow for Secure Distributed Applications

by

Winnie Wing-Yee Cheng

M.S., Stanford University (2004)
B. A. Sc., University of British Columbia (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author . . . .
Department of Electrical Engineering and Computer Science
August 3, 2009

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

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Information Flow for Secure Distributed Applications

by

Winnie Wing-Yee Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on August 3, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Private and confidential information is increasingly stored online and increasingly being exposed due to human errors as well as malicious attacks. Information leaks threaten confidentiality, lead to lawsuits, damage enterprise reputations, and cost billion of dollars. While distributed computing architectures provide data and service integration, they also create information flow control problems due to the interaction complexity among service providers. A main problem is the lack of an appropriate programming model to capture expected information flow behaviors in these large distributed software infrastructures. This research tackles this problem by proposing a programming methodology and enforcement platform for application developers to protect and share their sensitive data.

We introduce *Aeolus*, a new platform intended to make it easier to build distributed applications that avoid the unauthorized release of information. The Aeolus security model is based on information flow control but differs from previous work in ways that we believe make it easier to use and understand. In addition, Aeolus provides a number of new mechanisms (anonymous closures, compound tags, boxes, and shared volatile state) to ease the job of writing applications. This thesis provides examples to show how Aeolus features support secure distributed applications. It describes the system design issues and solutions in designing a prototype implementation and presents performance results that show our platform has low overhead.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

Acknowledgments

To me, this is the most important section of my thesis. Thinking back to all the years that I have been in graduate school, I am very fortunate to have had the opportunity to interact with great minds and truly inspirational figures. My advisor, Prof. Barbara Liskov, is a brilliant computer scientist and I admire her dedication to excellence in her work. She is also an influential mentor teaching me how to approach complex problems and distill them into important fundamental questions and to present ideas in simple and concise form. My co-advisor, Prof. Liuba Shrira, has also guided me in much of my thesis work and has been a constant source of encouragement. I am amazed at the breadth of her knowledge of related work and appreciated her insightful questions during our discussions that allowed me to step back and think about how this work fits into the greater scheme of system security. I would also like to thank my thesis committee, Prof. Sam Madden and Prof. Robert Morris, for their feedback on my work and the many good questions they raised to make this a better thesis.

I would like to acknowledge the wonderful present and past members of the Programming Methodology Group who have helped with my papers and presentations and created an environment conducive to learning: James Cowling, David Schultz, Dan Ports, Evan Jones, Dan Myers, Ben Vandiver, Ben Leong and Dorothy Curtis.

There are also researchers that have taught me a lot during my internships at HP Labs. Dr. Jun Li has been a great mentor and taught me many things about .NET and web services. Dr. Alan H. Karp introduced me to capabilities systems and I have enjoyed learning from his distributed systems drop-in sessions. I would also like to thank the Digital Printing and Imaging Lab for making me a better researcher.

On a more personal note, family and friends have been instrumental in making graduate studies an enjoyable experience. First, I sincerely thank my parents for their financial and emotional support since I came into this world. I would also like to thank Lykomidis Mastroleon for encouraging me to apply for the PhD program. Daniel R. Johnson has helped me tremendously in staying focused on my thesis research and has taught me many life skills outside the laboratories for which I am very grateful. I thank my sisters Annie and Linda for their 'care' packages before deadlines and prior to my defense. Friends from CSAIL, GW6/Leaaders-in-Life, Project ORCA and Ashdown House have made my years at MIT memorable. I had lots of fun cooking for Graduate Student Lunch (GSL) with Angelina Lee, Yuan K. Shen and Albert Huang. It was a blast building an autonomous submarine with Joshua Apgar, Ara Knaian, Sam Kenyon and several others. Jim Sukha and Grace Chau helped me with my RQE preparations. Ying Zhang and Shuodan Chen were always there to lend a listening ear. Manas Mittal and Manu Gupta were friendly floormates that wandered the halls of Ashdown House after sunset. Thanks to many others that I have not mentioned here but whose support and friendship will not be forgotten.

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

Chapter 1

Introduction

Private and confidential information, e.g., credit card numbers and medical records, is increasingly being stored online. Also increasingly this information is exposed due to human errors as well as malicious attacks.

There are many examples of information leaks. Rudder was touted as a convenient web-based financial planning tool where users can link their bank and credit card accounts all in one place [32]. However, an announcement in May 2009 [19] reported that their software had inadvertently shown users each other's bank account information. Ryerson University in Canada violated the Freedom of Information and Protection of Privacy Act with a software glitch that listed student names, ID numbers and grades on its web site and has been publicly criticized for their negligence [49]. WellPoint, a health insurer, exposed 130,000 of its customers' protected health information and personal records online [21]. There are numerous other documented cases [48] where software errors have jeopardized users' identity (e.g., Virginia Bureau of Insurance, Comcast, Automatic Data Processing, University of Virginia), personal financial data (e.g., Citigroup/ABN Amro Mortgage, CompuCredit, City of Riverside California), and patient health records (e.g., Ohio State University Medical Center, Georgetown University Hospital). These online systems are useful but for users to have confidence in using them, the security of online information must be addressed.

This thesis describes *Aeolus*, a new platform intended to make it easier to build applications that avoid the unauthorized release of information. The Aeolus security model is based on information flow control. Traditionally, online information has been secured through access control [15], which constrains who can read information, but not what can be done with the information. Information flow control is a complementary technique that instead restricts what can be done with information. For example, an administrative assistant in a medical clinic needs to know who the patients are in order to schedule appointments. If security is provided by access control, nothing prevents the administrator from then leaking the information. Information flow control allows the access while preventing the administrator from using the system to leak the information, e.g., sending all patient files in email.

Information flow control has been of interest in military systems [58], where there is a rigid classification of information, e.g., "secret" or "top-secret". Recently more flexible forms of information flow control have been proposed in which individuals are able to use discretionary control over their own information in a fine-grained way. As discussed further in Chapter 8, this recent work provides information flow control either through the use of special programming languages (e.g., [44, 51]), or at the operating system level (e.g., [18, 69, 35, 70]).

Although this recent work is valuable, we believe it does not provide the support that programmers need to build large-scale distributed systems that support information flow control. The operating systems work helps facilitate retrofitting legacy systems with support for information flow control, but it is lower level than what programmers want when building new systems. The work on security-typed languages can provide strong assurances and provable security properties; however, this approach requires new languages and complex type systems.

In contrast, Aeolus is defined at an intermediate level higher than the operating system, but not requiring the use of a new programming language. Our goal is to provide a tool that is easy to use and understand, yet provides the needed expressive power so that programmers can implement applications in a convenient way. To this end, we have focused on developing a model that is both simple and expressive: the model provides the primitives that we believe will make it easier to write secure applications with information flow control. Our approach allows implementations to be distributed and to make use of components developed by third parties.

An important way in which Aeolus simplifies application implementation is by embedding its security model within a distributed model of computation. This model is object-oriented: a distributed program is composed of objects residing at one or more machines. These objects are only able to communicate via a secure file system, or by making remote procedure calls to one another's methods. Internally objects run multiple processes; each time a call arrives it runs in its own process. These processes are isolated: they do not share memory directly. However, Aeolus provides a limited way for them to communicate, through secure shared state.

A secure distributed model of computation has not been provided by any of the other approaches.

Aeolus is implemented as a platform that runs on a collection of machines within an Aeolus configuration. All user code running on those machines runs on top of the platform. The platform supports the Aeolus programming model. It tracks information as it flows within and between the machines that make up the configuration, and it controls the movement of that information, both among members of the configuration, and to the outside (to machines that are not in the configuration and to I/O devices).

Similar to the earlier work, Aeolus focuses on controlling information that is communicated directly, for example, written in an email message. Aeolus is aimed at preventing errors from undermining information security, rather than malicious attacks. It does not address leaks through covert channels, although we have been careful to avoid introducing additional opportunities for covert channels via our mechanisms.

1.1 Contributions

This thesis makes a number of contributions.

1.1.1 The Model

We provide a new programming model in which application developers can specify how sensitive data can flow through a distributed application composed of modules from a diverse set of providers. Aeolus allows a developer to put together an application with components that run on many different machines and that may have been produced by different organizations; nevertheless the programmer can ensure that sensitive data is treated properly, i.e., according to the desires of the owners of the data.

The Aeolus model is based on concepts defined by others, in particular principals, tags (categories), and labels, and uses information flow rules to enforce both confidentiality and integrity, in particular, requiring authority or privilege to declassify or endorse. However, it combines these concepts in a new way that we believe makes the security model easier to use and understand.

Fundamental to flow control is the use of authority. In Aeolus, authority resides with principals and processes always run on behalf of a principal. Aeolus only allows a process with the proper authority to release confidential information or to vouch for information having a certain integrity. Aeolus provides support for both information flow control and access control, but using separate mechanism. In particular, it supports access control in the usual way, using principals and access control lists.

1.1.2 Support for Modular Program Construction

A distributed computation is made up of software modules from different developers and sometimes from third parties. It is crucial to effectively control the dynamic behavior of different modules in these large-scale applications. Most important is the support for a programming methodology where most code of an application runs with minimum privileges to prevent programming errors from causing a costly information leak. If only small portions of code perform privileged operations, they can be easily isolated and verified. Then security of the application can be guaranteed by analyzing only these critical modules rather than by examining the entire code base.

Aeolus provides mechanisms that explicitly address modularity. It provides this support through three complementary mechanisms.

First, Aeolus provides explicit support for the principle-of-least-privilege [52]. It is well understood that this principle is the basis for writing secure systems. Aeolus provides mechanisms that can be used to cause code to run with minimal or no authority. The latter way of doing things is particularly useful, because in this case the code will be unable to compromise security even if it has errors.

The second mechanism is the ability to bind authority to code, through the use of anonymous authority closures. For example, a module that checks passwords can be granted authority to produce the one bit `isValid` result. Before granting the authority, a user can inspect this code and verify that it deserves the authority, the amount of information being leaked in its results is small enough to be acceptable. Aeolus ensures that the authority cannot be usurped: it is granted to just the closure and cannot be used to do something different, for example, to print the password on the printer. Additionally, Aeolus allows dynamic delegation and revocation of authority to the closure, and it provides support for software upgrades.

Third, Aeolus makes the use of authority explicit and applies a *just-in-time* methodology. Modules that are able to run with authority have to indicate exactly when they want to use it. For example, the password checker contains code to do the checking, but it can be written so that the release of the answer, which requires authority, happens only at the end. This provides a proof point: a point at which one can reason about whether it is safe to release the information. Any inadvertent release of the password data prior to this point is not possible.

1.1.3 New Mechanisms

Aeolus provides a number of new mechanisms that make it easy for applications to accomplish their tasks securely:

- **Distributed Object Model**. Distributed programs running on the Aeolus platform are composed of a collection of objects that we refer to as *app-objects*.

App-objects can communicate only through remote procedure calls to one another's methods, or through the secure file system. App-objects run as a collection of processes, and these processes are isolated, except that they can communicate via shared state as discussed below.

- **Anonymous Authority Closures**. Anonymous authority closures were already discussed above; we expect them to be heavily used in applications. Authority closures tie authority to code so that it cannot be misappropriated. At the same time, they support software upgrades and dynamic delegation and revocation of authority.

- **Boxes**. All information flow mechanisms track contamination and processes become contaminated when they read contaminated information. Furthermore, once a process is contaminated, it requires authority to remove that taint. Boxes allow contaminated information to be exchanged safely while providing senders and receivers the means to control when they become contaminated. For example, a box can be used as an argument in a remote procedure call, and the recipient can protect itself from becoming contaminated by the box's content if all it is doing is passing the box on to a third party, without looking at the information inside the box.

- **Shared State**. We support fine-grained sharing of volatile information between concurrent processes while tracking information flow using a special shared state mechanism. Processes can share objects in the shared state; additionally, shared state provides a means for processes to synchronize. However, the use of shared state is limited to processes within the same app-object: each app-object has its own shared state.

- **Compound Tags**. In many systems, the tags used to compartmentalize data are related. For example, in a medical clinic, each patient's data has its own tag, but all of these tags are related in that they represent information about patients at the clinic. Compound tags allow relationships among tags to be captured and permit efficient delegation of authority for groups of related tags.

1.1.4 Proof-of-Concept Platform

Application developers can construct secure programs using the Aeolus model. We demonstrate that our model is implementable by developing a distributed computing platform to deploy these programs. In particular, we provide a platform for running local and remote software supporting local processes as well as remote ones (e.g., web services). Our working prototype demonstrates that our proposed programming model can be implemented efficiently, in ~20,000 lines of code, while supporting the functionality needed to run real applications.

1.1.5 Application Use Cases

We also have application case studies that show that our model is sufficiently expressive to support complex program interactions. Although the main goal for Aeolus is to support new applications, we also looked at its use to prevent security errors in an existing application. These studies provide a proof of concept for Aeolus, allowing us to argue that our model is complete.

1.2 Outline

The remainder of this thesis is structured as follows. The next chapter provides an overview of Aeolus defining our threat model and our high-level application model. Chapter 3 describes the Aeolus programming model and it is followed by examples that use our programming mechanisms in Chapter 4. Chapter 5 presents the design of our distributed computing platform. The evaluation of Aeolus is broken into two chapters: Chapter 6 examines the performance of the Aeolus platform and Chapter 7 applies the Aeolus model to several application scenarios to evaluate its expressive power. Chapter 8 compares our research with other related work. Finally, Chapter 9 concludes and discusses extensions to this work.

Chapter 2

Aeolus Architecture

This chapter describes the system environment we assume for Aeolus, and the application model it presents to users. We also discuss our threat model.

2.1 Aeolus Configurations

Aeolus provides information flow control for a distributed computing environment. Our system consists of one or more nodes. We assume nodes are allowed to enter the system through some kind of admission control, e.g., a node that wants to join must present a certificate from a trusted party. We assume this party uses some mechanism, beyond the scope of our system, to determine whether it is appropriate to allow a node to join. Every system member has some associated registration information known to all other members, including an IP address and a public key.

Our platform runs on all nodes in the system. We assume all in-system nodes are trusted to run our platform and to ensure that user code is running on top of our platform. User code runs on behalf of a principal. Aeolus vests authority in principals, which it represents as principal IDs. However, we do not require that the way we represent principals needs to match how this is done on a node in our system. Instead, system nodes merely need to map from their representation to ours.

Aeolus tracks information flow within each node in the system and between system nodes. It allows sensitive information to flow in messages, but the messages are encrypted so that secrecy and integrity are protected. Aeolus also controls the flow of information between the inside and the outside of our system. This flow can take the form of communication with outside nodes. It can also involve the use of I/O devices: all I/O devices are considered to be outside the system. If information is flowing to the outside, the flow is allowed only if the application trying to send information has the authority to declassify it (i.e., remove its contamination). Similarly, information coming in from the outside has no integrity, which implies that the application receiving the information must have authority to endorse it if needed (e.g., verify the input prior to storing it in a file).

Similar to other systems that provide information flow control, Aeolus tracks information as it flows through programs, and determines whether programs have the authority to perform certain security-sensitive operations, such as releasing secret data. The latter operations require some way to track who has authority for what. In Aeolus, this is done through *authority state*, which is maintained by our platform. The authority state is logically centralized (although the implementation could be distributed).

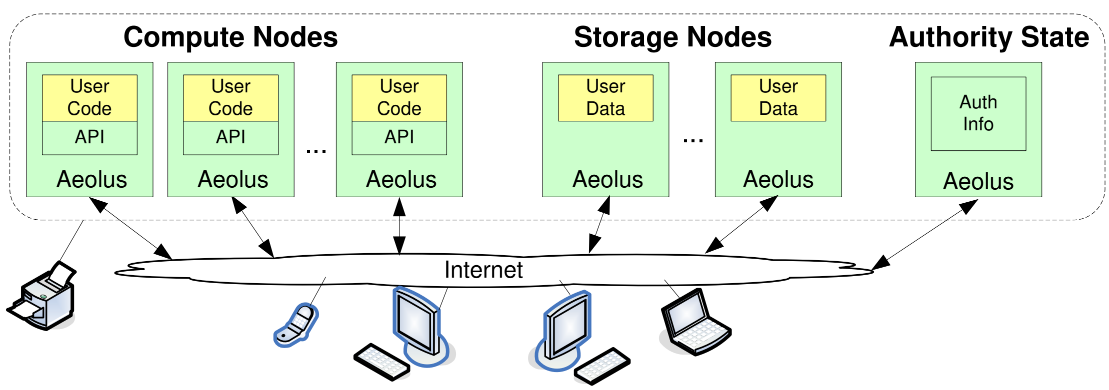

Figure 2-1: Aeolus Distributed Computing Environment

This architecture is illustrated in Figure 2-1. This figure shows an Aeolus system configuration with some nodes inside the system and others outside; additionally the devices are outside. The figure also shows that each node inside the system runs the Aeolus platform, which determines whether security-sensitive operations can be performed by consulting the authority state. As mentioned earlier, all user code running on an inside node runs on top of the Aeolus platform. The authority state is shown as being stored at a single node, which is how we handle it in our implementation.

This figure shows that some nodes are intended to run programs, while others are used only to store sensitive data; of course some nodes might be used both to compute and to store data.

2.2 Threat Model

Aeolus is aimed at preventing programming errors from undermining information security. The complexity of software and distributed nature of today's software make it difficult to ensure that the million lines of code in an implementation will not leak sensitive application information. Software often has bugs and systems can be configured incorrectly. We focus on software design methodologies and a programming model that can minimize the occurrence of such errors. We do not assume perfect confinement, e.g., a person can copy down displayed data; however, we assume that information leak happens inadvertently rather than through malicious attacks.

We trust that nodes within our system run our platform and system administrators responsible for these nodes will take actions to protect them from malicious attacks (e.g., network firewall, software patches, virus scans). We assume secure user password management and un-compromised authentication services.

A closure binds code to authority. However, to support software upgrades for closures, we accept new versions of this code, provided they are accompanied by a certificate that covers the code; we verify this certificate by using a public key associated with the closure. We assume that versions of closure code are supplied by a trusted code repository that manages the secret keys securely.

In addition, in our prototype, we implement the platform code on top of the language runtime, so this is also included in our secure base.

We focus on two types of data security issues: *confidentiality* and *integrity*. Confidentiality ensures that secrets cannot leak. Integrity ensures that information is vouched for properly. We do not address security issues related to the availability of data (e.g., denial-of-service attacks) and resource allocation of system nodes.

Aeolus does not address leaks through covert channels, although we have been careful to avoid introducing additional opportunities for covert channels via our mechanisms.

An Aeolus configuration might be used for a single distributed application, e.g., a medical information system running on many machines. It might also be used to run several disjoint applications; each of these might be itself running on several in-system nodes. All applications running on Aeolus must trust our secure base. However, disjoint applications do not need to trust the user code running on the nodes that support other applications; Aeolus will ensure that those applications cannot interfere with one another. Instead applications need only trust the user code that they make use of and even then, only if they give it some of their authority.

2.3 Application Model

This section provides an overview of how applications that run on Aeolus are structured.

We provide an object-oriented architecture in which an application is implemented by a number of *app-objects*. For example, a developer can divide a server application into app-objects for different tasks (e.g., one for handling incoming customer requests, one for retrieving product catalog data, and another for periodically computing statistics over usage logs).

Each app-object resides at a single node. An application may consist of app-objects residing all on the same node or on different nodes within our system. App-objects are isolated from one another: there is no direct sharing of data between them.

Each app-object provides methods that can be called by other app-objects to communicate with them. App-objects communicate via remote procedure calls. They may also receive calls from nodes outside of Aeolus, and can make calls to such nodes.

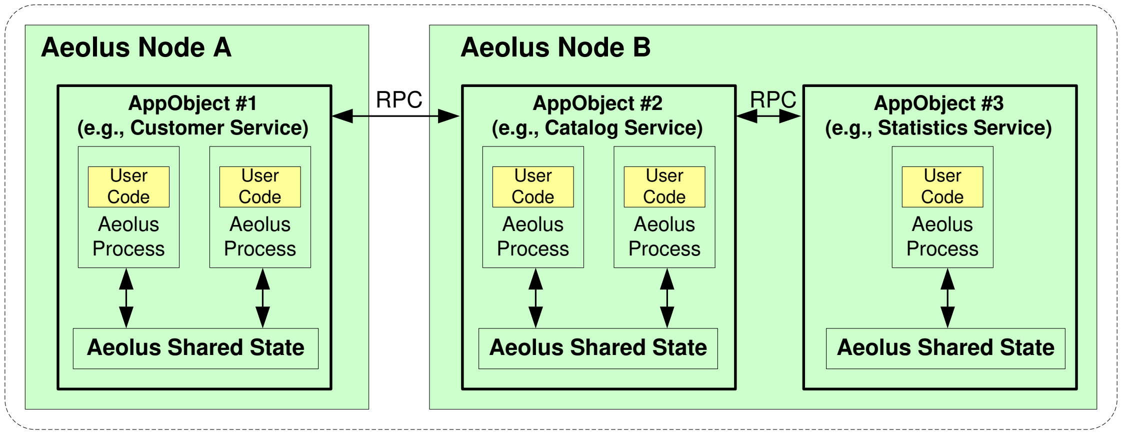

Figure 2-2: Aeolus Application Model

Within an app-object, there can be multiple processes. Some of these processes handle remote calls: each call runs in its own process. Others can carry out background activities. All of these processes are isolated: each runs in its own separate address space. However, Aeolus provides a limited way for processes within an app-object to communicate and synchronize via a *shared state* mechanism. Each app-object has its own shared state, which can be used only by the processes in that app-object.

This architecture is illustrated in Figure 2-2. This figure shows an example of an online store application running on Aeolus. The application is implemented using three app-objects: customer service, catalog service, and statistics service. The customer service app-object is deployed on a different node to offload the processing of incoming web requests. Our platform tracks information flow between Aeolus processes, across app-objects on the same node, and over remote calls to app-objects on other nodes.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Programming Model

The Aeolus model enforces information security by allowing application developers to label critical data and permits information to flow from one system entity to another only under certain conditions. Authority is needed to perform security-sensitive operations.

In this chapter, we describe the programming model and how applications execute under the constraints of our model.

3.1 Motivating Examples

First, we present some examples where information flow control is an important consideration in the design of applications.

Bob and the Tax Preparer. In this example, which is borrowed from [45], a vendor provides a service that, given a user's tax information (e.g., Bob's information), produces a tax form. The tax information and the resulting tax form are both confidential and should be visible only to Bob. To produce the tax form the service makes use of a proprietary database that should only be visible to it, and the resulting tax form is contaminated by this information. The system needs to provide a way for the tax preparer program to decontaminate the result so that Bob can see it, but also prevent the tax preparer program from making Bob's private information visible to third parties.

Sales Analysis. This example concerns the commercial sector where it is common to use outsourced services; a discussion of control of information in this environment is discussed in [27]. A third party produces a tool that analyzes sales information from many companies and produces results about customer preferences. Companies that use the tool have determined through some mechanism that the final result is acceptable to them (e.g., it doesn't expose details of their organizations). The tool needs a way to produce the final result so that each company using the tool can see the result. But additionally, the companies need a guarantee that the third party won't be able to expose their private sales information.

The Medical Clinic. Clearly confidentiality in medical information systems is a matter of great concern, e.g., see [41]. Here, we mention a couple of examples (in addition to the one discussed earlier concerning the administrative assistant). First, billing must be done based on the treatments and appointments that individuals had during the billing period. The bills need to be sent to the appropriate parties (the patient or his insurance company) but neither the raw data nor the bills should be otherwise exposed. Second, a statistics package requires access to all patient records; it is trusted to produce a result that obfuscates the data but it needs a way make this public.

3.2 Principals, Tags and Labels

The Aeolus model is based on three key concepts: principals, tags, and labels. *Principals* represent users or roles (e.g., user Alice or the company contractor role). Application code runs within processes. Every process in Aeolus is associated with a principal and runs with the authority of this principal. This allows it to perform certain privileged label manipulations and delegations as discussed below.

Tags provide a way for principals to categorize their information. For example, Bob might define three categories, one for public information, one for family information, and one for private information.

Both principals and tags are represented by opaque randomly-generated and globally-unique identifiers, referred to as *principal identifiers (PIDs)* and *tag identifiers (TIDs)*, respectively.

Labels are sets of tags and are used to control information flow. Aeolus allows certain objects to be labeled. All data objects (such as files and also objects in the shared state) and all processes have two labels: a secrecy label, L_S, which reflects confidentiality of information, and an integrity label, L_I, which reflects the integrity or validity, of information.

A tag can be used in a secrecy label or an integrity label or both, as shown in Figure 3-1. For example, an application developer may use our model to create a tag to represent the `AliceShopping` sensitivity category and include it in the secrecy label to restrict where data derived from Alice's transaction history can go. The same `AliceShopping` tag is used in the integrity labels of files containing online purchases confirmed by Alice.

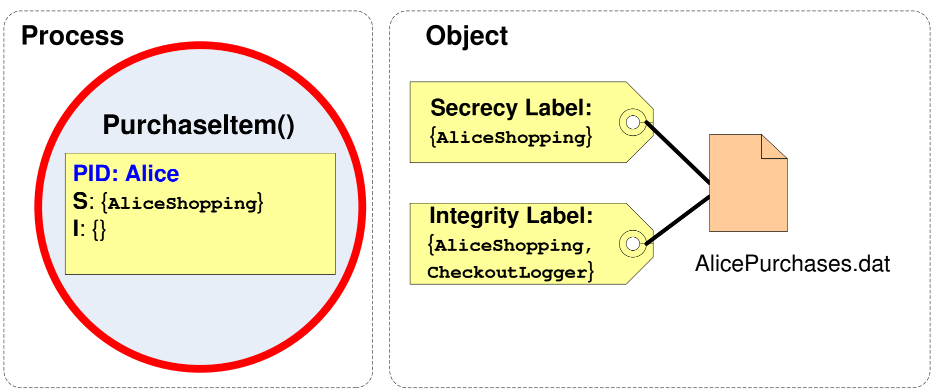

Figure 3-1: Process, Object, Tags and Labels

When a labeled object is created, it is assigned a secrecy label and an integrity label. These labels are immutable throughout the lifetime of the object. As a process executes, it may access sensitive information from different sources and as it does so its labels change. For example, an application developer might create two files: `FileA.txt` and `FileB.txt`. `FileA.txt` is used to store Alice's personal data and is assigned the tag `AlicePersonal` in its labels while `FileB.txt` is used for Bob's data and is assigned the tag `BobPersonal`. The label of a process that reads `FileA.txt` will contain the `AlicePersonal` tag to reflect the contamination by Alice's personal data. Next, we discuss the precise rules that govern such information flows.

3.3 Information Flow Control Rules

The information flow rules are defined as the conditions that must be satisfied for information to flow from a source A to a target B. For example, this can happen when process A writes to object B or when process B reads from object A. There are two conditions and both of them must be satisfied. These rules are similar in concept to the conventional lattice-based rules defined in [14].

Information Flow Constraints on Source A and Target B:

- Secrecy Condition: $A.L_S \subseteq B.L_S$
- Integrity Condition: $A.L_I \supseteq B.L_I$

When these conditions are satisfied, we say that A's labels are *no more restrictive* than B's.

A label L_A is a subset of another label L_B if and only if L_B contains all the tags in L_A. The secrecy condition says that the secrecy label of the source of the information, A, must be a subset of the secrecy label of the target, B. The integrity condition says that the integrity label of the target of the information, B, must be a subset of the integrity label of the source, A (or equivalently, the integrity label of A must be a superset of the integrity label of B).

The secrecy condition ensures that confidentiality is maintained as data propagates. For example, if a process has knowledge of Company A's sales information, that is, its secrecy label contains the corresponding tag, then the process can only write to a file with secrecy label containing at least this tag. Thus, the secrecy condition allows a process with secrecy label {`CompanyASales`} to write to a file with {`CompanyASales`, `CompanyBSales`} but not to one with {`CompanyBSales`}. The integrity condition prevents influences from low-integrity entities. For example, a file containing a set of verified account numbers may have the integrity label {`AccountingVerified`} and the integrity condition will not allow a process without this tag in its integrity label, for example, a process with an empty integrity label, to modify this file.

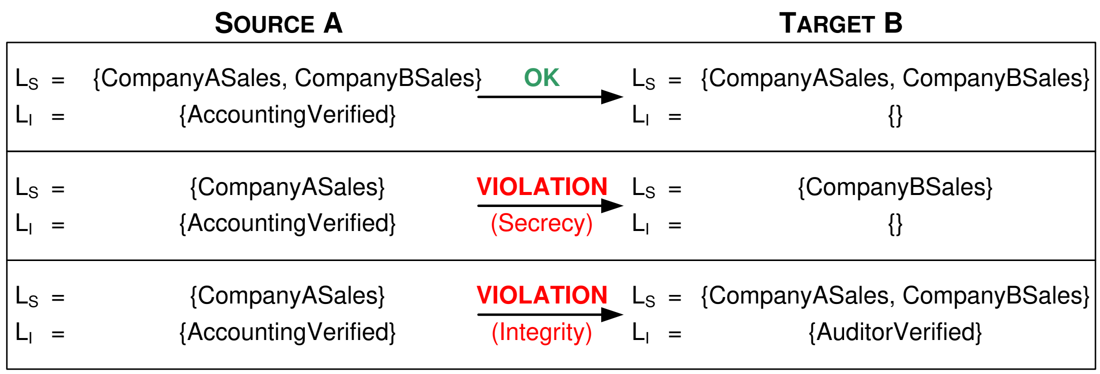

Figure 3-2: Examples of Information Flow Checks

Figure 3-2 illustrates these examples in greater detail. A violation in either the secrecy condition or the integrity condition will trigger an exception in our model.

3.4 Label Manipulations

To satisfy the information flow rules, processes sometimes need to manipulate their labels. Some manipulations are safe, while others are not.

Any process can perform safe label manipulations.

Safe Label Manipulations:

- `AddSecrecy(in T)`: Adds a tag `T` to the process' secrecy label.
- `RemoveIntegrity(in T)`: Removes a tag `T` from the process' integrity label.

These two manipulations are safe since they can only further constrain the use of data. For example, a process with the added tag in its secrecy label {`T`} can no longer write to a file marked with an empty secrecy label. These safe label manipulations preserve secrecy (i.e., sensitive information doesn't become unmarked) and do not jeopardize integrity (i.e., integrity can only be lowered).

On the other hand, the following two label manipulations are security-sensitive operations.

Privileged Label Manipulations:

- `Declassify(in T)`: Removes a tag `T` from the process' secrecy label.

- `Endorse(in T)`: Adds a tag `T` to the process' integrity label.

Removing tags from a secrecy label potentially allows confidential data to leave the system; since this can cause sensitive information to leak, this is considered a privileged operation. Tags in integrity labels indicate the degree of confidence, so adding tags to an integrity label must also be verified by the Aeolus Platform.

Aeolus ensures that the process has sufficient privilege before performing the operation. Our system allows a privileged label manipulation only if the principal of the process executing the operation *has authority for* the tag in question, i.e., the tag being removed in a declassification operation, or being added in an endorsement operation. Authority is discussed in Section 3.5.

Our model requires that *all* label manipulations be done *explicitly*. This is especially important for privileged label manipulations to be made explicitly because it prevents accidental and unintended flows and encourages a *just-in-time* methodology in which authority is used just at the moment that an intended flow happens. For example, the sales analyzer would remove the company tags at the last minute as it returns the result of its analysis; this reduces the chance that errors will cause release of sensitive information.

3.5 Authority

A principal that is authoritative for a tag can perform privileged label manipulations involving that tag. Authority starts with tag creation; when a process creates a tag, its principal is granted authority for that tag. Authority is extended both *implicitly* and *explicitly*. Implicit authority is obtained through the *principal hierarchy* and explicit authority is given through authority grants. We give examples in Chapter 4 of how implicit and explicit authority are useful in a medical clinic.

3.5.1 Principals

Principals are arranged in a *principal hierarchy*, allowing one principal to *act-for* another [20]. If a principal P1 is recorded as acting-for another principal P2, then P1 has all the authority of P2. The act-for relationship is transitive. The principal hierarchy is useful to capture organization structure (groups) and also it allows individuals to use different principals for different purposes (roles).

A process running as principal P can create a new principal. This is done with a `CreatePrincipal` request that returns a new principal ID, P1. In doing so, the creator principal P automatically acts-for the created principal P1.

Creating a new principal:

- `CreatePrincipal(out P1)`: Creates a new principal `P1`; the creator principal acts-for `P1`.

A principal P1 can allow another principal P2 to act for it by issuing an `ActFor(in P1, in P2)` request. This call is legal provided it does not cause a cycle in the principal hierarchy and the process' principal is authoritative for the actee principal P1.

Delegating `ActFor` authority:

- `ActFor(in P1, in P2)`: Allows principal `P2` to act-for `P1`.

3.5.2 Tags and Grants

Any principal can create a tag by issuing a `CreateTag` request on our platform.

Creating a new tag:

- `CreateTag(out T)`: Creates a new tag `T`; the creator principal has authority for tag `T`.

A new tag T is returned and the principal that issued the request is automatically given authority for the new tag T.

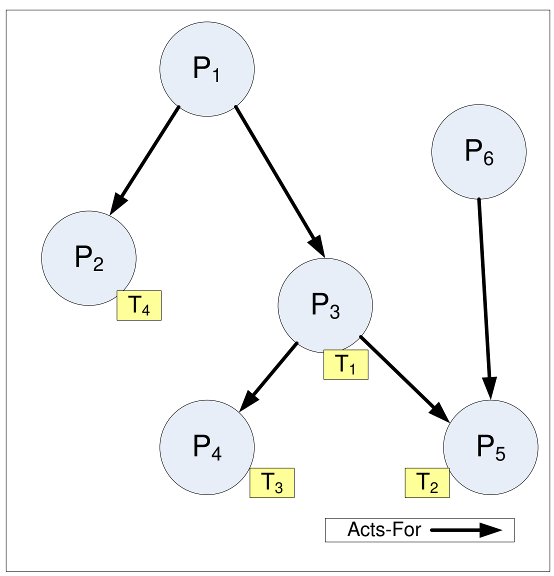

Figure 3-3: Act-for Relationships in Principal Hierarchy

Figure 3-3 shows an example of a principal hierarchy. The principal hierarchy is a directed acyclic graph with nodes representing principals and with edges pointed from an *actor* principal to an *actee* principal. This figure shows the principals that created the various tags; for example, principal P_4 created the tag T_3. The act-for relationships in the principal hierarchy allow P_1 and P_3 to derive authority for tag T_3

even though they did not create the tag. (Note that the act-for relationships should not cause loops as a loop will imply that all principals along the path should have the same derived authority and hence are better represented as a single principal. This single principal will represent a group role where all members have equal authority.)

With `ActFor`, the actor principal P2 implicitly inherits *all* the authority of the actee. Additionally we allow explicit delegation of authority for tags. Explicit delegation of authority is much more controlled and therefore safer since the grantor doesn't provide all the privilege. This method of delegation provides fine-grained control. Just as tags allow applications to categorize information and provide separate controls for different categories, grants allow applications to control authority over those categories in a constrained way.

When principal P1 uses `Delegate(in T, in P1, in P2)` to explicitly grant the authority for a particular tag T to principal P2, this operation is permitted if the process' principal acts-for principal P1 and principal P1 has authority for tag T.

Explicitly granting authority:

- `Delegate(in T, in P1, in P2)`: Grants authority for tag `T` from principal `P1` to `P2`

Figure 3-4 compares implicit and explicit delegation; in both cases, P_1 has authority for tag T_1. On the left, P_1 has been allowed to act-for P_3; P_1 thus has derived authority for tags T_1 and T_2 through principal P_3. If principal P_3 wants to limit the delegation to only tag T_1, it can do this by issuing `Delegate(`T_1`, `P_3`, `P_1`)` instead as seen in the right subfigure. An authority chain is maintained for each explicit delegation to keep track of the origin of the authority.

3.5.3 Compound Tags and Static Grouping

Explicit delegations can be cumbersome to use. For example, a clinic administrator has to delegate the tags of each patient to the principal generating the billing

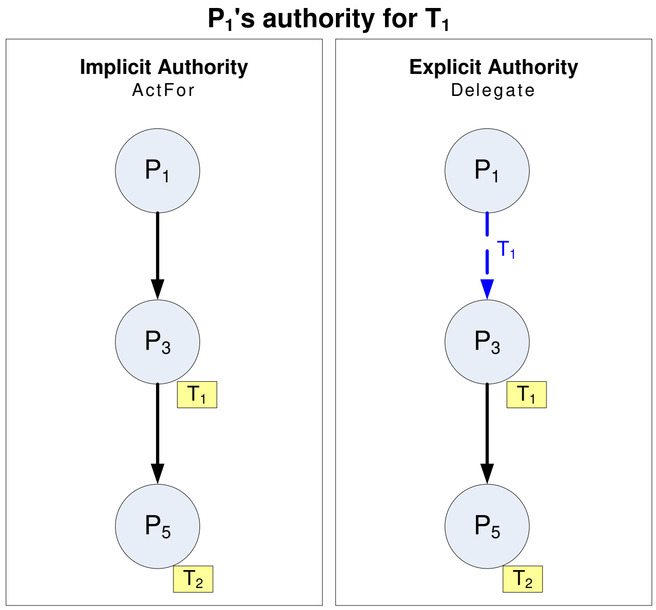

Figure 3-4: Comparing Implicit and Explicit Delegation

information.

To support this common usage pattern, we provide *compound tags*. This mechanism allows tags to be grouped *statically*, as they are created, e.g., all patient tags are created in the same group. The compound tag can be delegated to a principal and this gives this principal authority for all tags in the group.

A compound tag (or top-level tag) is created with the tag creation operations mentioned previously, namely, `CreateTag(out T)`. To create a sub-tag, a process issues `CreateSubTag(in T1, out T2)`, where `T1` is the top-level tag that the newly created sub-tag `T2` is associated with. When a principal is given authority for a compound tag, it has authority for all its sub-tags. `T1` must be a top-level tag; otherwise an exception is raised.

- `CreateSubTag(in T1, out T2)`: Creates a sub-tag `T2` of top-level tag `T1`; the principal that makes this call has authority for tag T2.

We chose to group tags statically (i.e. at creation) because this approach has significant implementation advantages over grouping them dynamically: as discussed in Chapter 5, we need not store static groups using explicit data structures. Furthermore the examples where large numbers of delegations were needed allowed a static solution: it was natural to think of patient tags as being related, or to think of all users of a web service as being related.

Static grouping doesn't work for every situation. For example, compound tags don't work for the sales analyzer, since in this case we cannot expect that the sales tags for the different companies are put into same group *a priori*.

We also chose a simple 2-level hierarchy for tags rather than a general hierarchy. We did this because there appeared to be no need for a general hierarchy in our examples. If we discover a need for a general hierarchy, the model can easily be extended to provide it.

3.5.4 Revocation

Aeolus allows authority to be revoked. Implicit authority can be revoked by removing a link in the principal hierarchy using `RevokeActFor(in P1, in P2)` where P1 is the actee principal and P2 is the actor principal. The removal has a transitive effect: not only is P2 no longer able to act-for P1, but all principals that act-for P2 no longer act-for P1. Explicit authority can be revoked by removing a delegation using `RevokeDelegate(in T, in P1, in P2)` where T is the tag (compound tag or subtag) for which authority was granted, P1 is the grantor principal and P2 is the grantee principal. This also has a transitive effect: if P1 delegated authority for tag T to P2, and later P1 revokes this delegation, this removes authority for tag T from P2 and also from any principal P2 granted that authority to using this authority. Our use of *delegation chains* allows revocation to match intuition: any delegations that happened as a result of the first delegation are also undone. For example, if Alice delegates tag T to Bob and Bob delegates to Tom, and then later Alice revokes Bob's authority for tag T, this takes away Tom's authority for tag T as well.

Revoking authority:

- `RevokeActFor(in P1, in P2)`: Revokes principal `P2`'s ability to act-for `P1`.
- `RevokeDelegate(in T, in P1, in P2)`: Revokes the delegation of tag `T` from principal `P1` to `P2`.

Similar to delegations, requests for revocations are checked to ensure that they are issued by a principal with sufficient authority. The principal of the process issuing the request must be authoritative for the actee principal P1 in `RevokeActFor(in P1, in P2)` and for the grantor principal P1 in `RevokeDelegate(in T, in P1, in P2)`.

3.5.5 Principal Hierarchy

The principal hierarchy is a directed acyclic graph that expresses act-for relationships between principals. It has the principal P_{root} at the top of the hierarchy and the P_{public} principal at the bottom. The P_{root} principal acts-for all principals but no principal acts-for it whereas the P_{public} principal does not act-for any principal and has no authority but all principals act-for it.

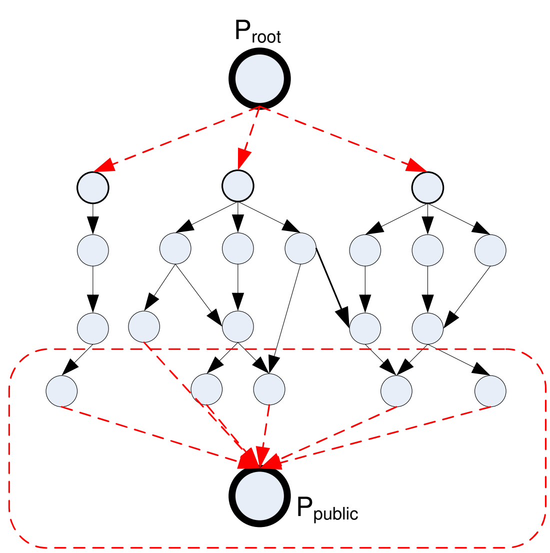

Figure 3-5: Special Principals

We provide a way to talk about the P_{public} principal explicitly. Any process can switch to running with the P_{public} principal. When a process is running with this principal, it doesn't have any authority and therefore cannot perform privileged operations such as declassifications and endorsements. Also, the system will not allow another principal to assign authority to P_{public} via explicit or implicit delegations. Furthermore, this principal cannot create tags and subprincipals. Running with the P_{public} principal is ideal for sections of code where we simply want to track the propagation of sensitive data with no need to perform privileged operations.

Our system also has a P_{root} principal. It is up to the application using the system to carefully control login so that this principal is used only when absolutely necessary. Aeolus prevents a process running as P_{root} to allow another user-created principal to act-for this principal.

3.6 Authority Closures

When a principal is given authority via delegations, there is no guarantee as to what code will use the authority. There are situations in which a user is willing to allow a service to declassify some of its data only after some transformations have sufficiently obfuscated the data. *Authority closures* allow programmers to obtain such a guarantee by granting authority directly to code. We expect authority closures to be widely used. For example, statistics over a medical database can be published provided details about individuals are not exposed. Similarly, a principal might be willing to endorse some data after a computation has checked that the data meets certain constraints. For example, an input verifier can check that a parameter is in the proper format and will not subject the program to an injection attack.

When an authority closure is created, it is associated with a new principal but that principal is anonymous and unrelated to any other principal. Since the anonymous principal is unrelated to any other, when the closure is first created it has no privileges. However, we allow authority for specific tags to be delegated to it, and also it can be made to act-for some (non-anonymous) principal. Importantly, though, there is no way to make some principal act-for the closure or for the closure to delegate authority for tags to another principal. We check and prevent an anonymous principal from issuing `Delegate` and `ActFor` requests. That way, we can guarantee that only the closure can use the authority granted to it. Limiting the closure to the authority prevents security leaks due to errors that happen later. For example, suppose the programmer who created the closure was able to act-for it; then later, he could accidentally misuse the closure's authority.

Conceptually, an authority closure binds an *anonymous* principal to some code provided when the closure is created. Binding directly to code provides very strong guarantees, but doing so is undesirable, for two reasons. First, we don't want Aeolus to be a repository for code, nor to be bound to code repositories, e.g., certain file systems where code is stored. This problem could be solved by storing a hash of the code rather than the code itself, but this doesn't solve the second problem: providing support for software upgrades.

To allow software upgrades, we instead bind a closure to a key that is provided when the closure is created. We assume here a trusted version management system used by the application; the key provides confidence that the version came from the trusted source. `CreateClosure(in key, out CL1)` takes in a `key` as an argument and returns a new closure ID `CL1`. During its creation, the `key` and a new anonymous principal are associated with this closure ID `CL1`. When an authority closure is invoked, the requester supplies the closure ID `CL1`, the code to run, and a certificate `cert`. This certificate must cover the code and must be signed by the closure's `key`; if this check fails, the call isn't allowed.

Here are some operations related to creating closures and delegating authority to them:

- `CreateClosure(in key, out CL1)`: Creates a new closure with the `key`, associates an anonymous principal `PA` with the closure, and returns the closure ID `CL1`.

- `ClosureActFor(in P1, in CL1)`: Adds an act-for link from the anonymous principal of closure `CL1` to actee principal `P1`, provided that the process' principal acts-for principal `P1`.

- `ClosureDelegate(in T, in P1, in CL1)`: Gives authority for tag `T` from grantor principal `P1` to grantee anonymous principal `PA` of closure `CL1`, provided that the process' principal acts-for principal `P1` and principal `P1` is authoritative for tag `T`.

An example of an authority closure is the method `IsPasswordValid(in username, in password, out isValid)` which checks against a file containing user and password information. The method returns a boolean value `isValid` on whether the input password is correct. The file is tagged by a tag $\mathtt{T}_{Password}$ in its secrecy label to protect passwords from leaking. In this case, a developer trusts that the output boolean value can be declassified since it reveals very little information about the actual passwords.[1] The `IsPasswordValid` method performs the privileged label manipulation, $\mathtt{Declassify(T}_{Password}\mathtt{)}$ just before returning the boolean value. The authority to remove this tag from the process' secrecy label is granted to just to this authority closure, which ensures that the authority can only be used when running this code.

We chose our solution of using a key because it gives maximum flexibility. Not only does it allow for software upgrades, it also allows for running different code on different machines (e.g., on .NET and on JVM). But still the system is secure, assuming a trusted code repository.

3.7 Execution

Once a process starts running, it can switch to run code as different principals via calls and forks. In addition, the developer can specify the declassifications and endorsements that are applied to the process' labels before running this code. Switching

[1] Note that this 1-bit output still reveals some relationships between users and passwords. Iterative invocations of this method can potentially be used in dictionary attacks. However, our goal is not to achieve perfect secrecy but merely to allow developers to identify code segments that are highly critical to system security.

to different principals allows code to run with only the privilege it needs for its task, following the principle-of-least-privilege. Applying declassifications and endorsements to specific code is an example of our just-in-time methodology.

3.7.1 Local Calls

Most local calls execute as-is and without the intervention of Aeolus. However, when the application wants to make a local call with a change in principal (e.g., to P_{public}), such requests must go through our system.

- `Call(in C, in P, in listS, in listI)`: Runs code `C` of type `AeolusCallable` in the same process with principal `P`. The process' labels are adjusted using the lists of tags `listS` and `listI` to apply declassification and endorsement, respectively, to the process' labels prior to invoking code C.

Code objects of type `AeolusCallable` have an `Invoke` method that our system can call. This method has neither arguments nor results; instead the object can hold arguments and return values in its internal state.

When a process issues a `Call`, it can specify the principal `P` that it wants to run code `C` with. It can also optionally specify the lists of tags `listS` and `listI` for applying declassification and endorsement to the process' labels, respectively. Aeolus checks that the process' principal acts-for principal `P` and that the caller principal has authority for all tags in `listS` and `listI`. The `Call` then runs with the adjusted process' labels (i.e., the tags in `listS` are removed from the secrecy label and those in `listI` are added to the integrity label). After the `Call` finishes, the caller's principal is restored and the caller process' labels are updated to reflect any contamination picked up as a result of the call (i.e., its secrecy label is union-ed with the caller's and its integrity label is intersected). The ability to change the labels at the same time that the principal is changed is important because the callee may not have the authority to do this itself. For example, a printer controller is run without any authority; it is passed the data to be printed as an argument and it must have a null secrecy label since otherwise it won't be able to use the printer.

3.7.2 Local Forks

A process can run as multiple threads, all in the same address space; in this case, all the threads have the same principal and labels. A `fork` can be used to create a new process that has its own address space and information flow state:

- `Fork(in C, in P, in listS, in listI)`: Runs code `C` of type `AeolusCallable` in a new process with principal `P`. The process' labels are adjusted using the lists of tags `listS` and `listI` to apply declassification and endorsement, respectively, to the process' labels prior to invoking code `C`.

Again code `C` is an object of type `AeolusCallable` with an `Invoke` method. When a process issues a `Fork`, it can run with the same principal as the caller's or it can optionally specify the principal `P` that it wants to run the code with. Aeolus performs the same information flow and authority checks as for a `Call`. If these checks pass, a new process is started with the appropriate principal and process' labels. Code object `C` is copied (i.e., serialized) to the new process. This code object may contain arguments and the copy of these arguments is the only data that the code can access. A fork is not expected to return to the caller so the caller's labels are unaffected by those of the forked process.

3.7.3 Example of Calls and Forks

The ability to limit the authority of code segments is an important property in constructing secure distributed applications. An application is a conglomeration of functions with different responsibilities. To minimize programming errors, we want to give functions only as much authority as they need to get the job done. Application developers are strongly recommended to identify critical functions and assign different principals to run various portions of their code and to carefully consider how authority changes can impact the overall information security of their programs. Forks and Calls allow different parts of code to run with different authority.

In Figure 3-6, we show an example of a Clinic Administrator application that uses `Fork` and `Call` to restrict the privilege when performing different tasks. An administrator may run this application on his/her desktop machine and use it to print out patient visit summary information at the end of a consultation, to look up phone numbers of a medical lab, or to send e-mails to hospital staff. The `Main` code presents a menu of tasks for the administrator to select from. In this example, the administrator wants to print patient Bob's visit summary and the application invokes the `PrintVisitSummary` code with Bob's patient ID. To avoid errors in mixing up patient records, the application forks off a new process to run this code and reduces the authority of the new process by running it with $P_{BobPatient}$ rather than the $P_{ClinicAdmin}$ principal that has authority for all patients' tags.

The `PrintVisitSummary` code uses the subroutine `GetConsultationSummary` to examine Bob's records and generate a summary. This subroutine is invoked using `Call` and is made to run with the P_{public} principal as this code does not require any authority. By running this code with P_{public} rather than $P_{BobPatient}$, the developer can be assured that programming errors within `GetConsultationSummary` cannot leak Bob's sensitive medical data. When this method accesses Bob's patient record, it adds the tag $T_{BobMedical}$ to the process' secrecy label and returns the visit summary. The `PrintVisitSummary` method can then exercise its authority to issue a `Declassify` prior to printing the patient summary on the local printer.

3.7.4 Authority Closures

Any process can invoke an authority closure.

Clinic Administration Application

`Main` code:
P=$P_{ClinicAdmin}$
1. Select a task (e.g., Print Visit Summary, Lookup Phone Directory, Send E-mails)
2. Choose to run `PrintVisitSummary` for patient Bob
2a. `PrintVisitSummary.PatientID` = Bob's ID
2b. `Fork(PrintVisitSummary,` $P_{BobPatient}$`, {}, {})`

`PrintVisitSummary.Invoke()` code :
P=$P_{BobPatient}$
1. Generate Consultation Summary
1a. `GetConsultationSummary.PatientID = this.PatientID`
1b. `Call(GetConsultationSummary,` P_{public}`, {}, {})`
1c. `summary` = `GetConsultationSummary.summary`
2. `Declassify(`$T_{BobMedical}$`)`
3. Print `summary` to local printer

`GetConsultationSummary.Invoke()` code:
P=P_{public}
1. Retrieve patient data and produce visit summary
1a. `AddSecrecy(`$T_{BobMedical}$`)`
1b. Read data with `this.PatientID`
2. Generates and returns `this.summary`

Figure 3-6: Example Usage of `Fork` and `Call`

- `CallClosure(in C, in listS, in listI)`: Forks a process to run code `C` of type `AeolusClosureCallable`, with the anonymous principal `PA` associated with the closure. The process' labels are adjusted using the lists of tags `listS` and `listI` to apply declassification and endorsement, respectively, to the process' labels prior to invoking code C.

When invoking an authority closure, code `C` is an object of type `AeolusClosureCallable`, which is a sub-type of `AeolusCallable` and hence, has an `Invoke` method that our system can call. In addition, the `AeolusClosureCallable` also includes a closure ID in its internal state and a method `GetCertificate` that can be used to obtain the certificate that proves the code is that of the closure.

When a `CallClosure` request is made, Aeolus uses the closure ID in Code `C` to retrieve the key and anonymous principal associated with the closure. It checks to ensure this code is certified by the key (by examining the certificate associated with `C`) before executing it with the anonymous principal. The requester can include lists of tags `listS` and `listI` to apply declassification and endorsement, respectively, to the process' labels prior to invoking code `C`. Aeolus checks that the caller's principal has authority for these tags. While the closure is running, the calling process is blocked waiting for it to complete. When the closure returns, the `AeolusClosureCallable` object is copied back to the caller's process, and the caller's labels are adjusted to reflect any contamination picked up as a result of the closure call.

While the closure is running, the system prevents the process from delegating its authority or allowing another principal to act-for it. Furthermore, the closure is executed in a new process and hence its authority cannot be usurped by threads running concurrently in the caller's process.

3.7.5 Remote Procedure Calls

Remote procedure calls are carried out as method invocations on remote objects. To a developer, a remote method is indistinguishable from a local method invocation. Our platform ensures that the method runs with the caller's principal and process labels on the remote machine and when the result is returned, the local process' labels are updated with the remote process' labels. Additionally, at the called object, the call runs in its own process, with its own address space and authority state. If the developer wants to run the remote call with a different principal or process labels, it can do so using `Call` and invoking the remote object within the `AeolusCallable` code object.

3.7.6 Launching app-objects

Any process can launch an app-object by calling:

- `Launch(in C, in P, in S, in I)`: Starts up a new app-object with configuration `C`. This app-object contains a single process, running with principal `P`, and with labels `S` and `I`.

A caller process launching an app-object must satisfy certain information flow constraints before the launcher will create this app-object. It supplies configuration `C` which contains all the necessary information (e.g., path to code libraries, input arguments, information about what code the app-objects process will run) for the launcher code to bootstrap the app-object. Since information can be leaked through this data directly or covertly, Aeolus requires the caller to have a null secrecy label. Similar to Calls and Forks, the caller process' principal must also act-for `P`. Furthermore, the `I` label must be no less restrictive than the caller process' since information flows from the caller process to the new process in the created app-object. Once the new process starts, it runs completely independently from the caller process; for example, the new app-object has its own shared state that the caller process cannot access.

3.7.7 Logging in

An app-object can also start running through a log-in procedure. We assume servers have an authentication mechanism that determines what principal is assigned when a user logs-in. Since servers within our system are trusted, different servers can use their own log-in techniques. We do not require uniformity in the way servers represent their principals; instead they only need to map their principals to ours.

These trusted log-in procedures present Aeolus with the principal ID of the logged-in user and the code to start running with. Aeolus creates an app-object to run the code in a process within this app-object. The log-in procedure can also specify the initial secrecy and integrity labels to run this process with.

3.8 Data

3.8.1 Files

Many applications require support for persistent data so that computation results and user data can be stored on disks and retrieved at a later time. Networked file systems and databases are typically used for these tasks. In addition to data persistence, they can also act as repositories for data that is shared by users on different machines. For example, an organization may have various application servers accessing a common set of files. A healthcare organization may have patient health records and personnel information stored over several storage servers. This thesis provides for secure persistent storage via a file system.

File systems represent data in a hierarchical manner. Directories are like containers and files are like objects inside containers. Containers may have on them descriptions about the objects they store. A container can also store other containers. As one opens a container, the objects revealed may be more sensitive than the outside. This is analogous to a sealed envelope with the address of the intended recipient written on it and within it, a confidential letter. Our model for persistent storage interfaces is based on this concept.

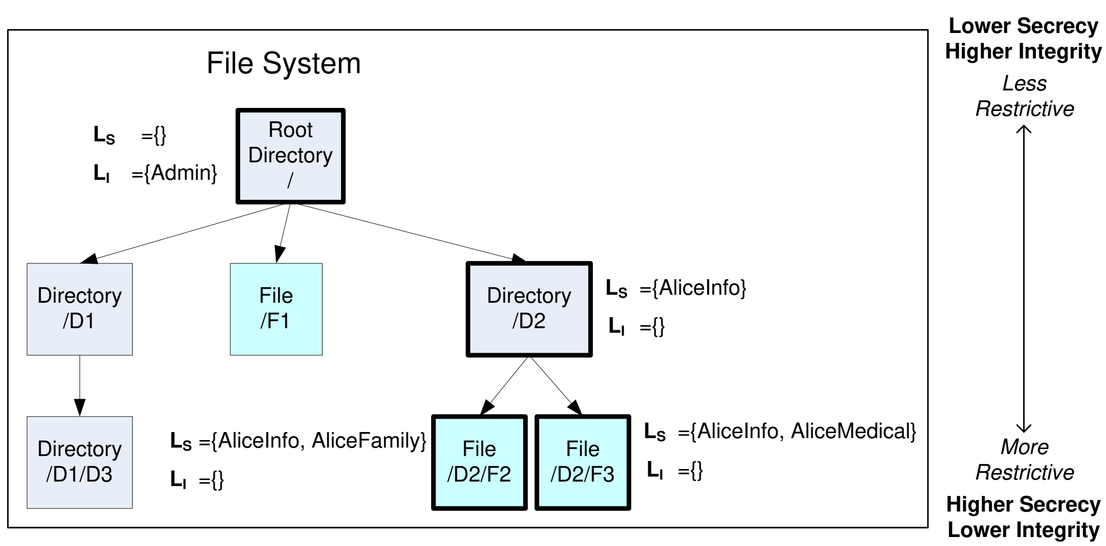

Figure 3-7: Labels on Files and Directories

Figure 3-7 shows how file systems are modeled using this container-object concept. The containers and objects higher up in the hierarchy are less restrictive. For confidentiality, this translates to more sensitive data being stored further down in the hierarchy, making them less accessible for reads and disclosures (e.g., putting Alice's patient records inside the "Alice" directory). For integrity, this means that it is harder for a process to have the authority to write to a container or to object higher up such as the root directory.

Directories are arranged in a hierarchical manner and directories and files exist logically within directories. Directories are special files with information about the files they contain. The labels on directories reflect the information flow concerns of the metadata (e.g., the existence of particular files). Files have immutable labels and a file's labels must be no less constraining than those of the directory that contains it. File reads and writes are allowed only if the information flow constraints are satisfied.

To create a file, the parent directory is updated (read and written) and the new file is created inside this directory. The parent directory contains the list of files and the new file is added to this list. This directory is also read because if the file already exists, this can trigger an exception and a process can learn about the existence of a file in this way. Therefore, the caller must have secrecy and integrity labels equal to the directory's to permit the update. However, the file to be created is often intended to be more restrictive than the parent directory and more restrictive labels can be specified for it. For example, the directory `\user\alice\` may have the secrecy label {`AliceInfo`}; Alice can create a file inside this directory with a process that is currently running with secrecy label {`AliceInfo`} and specify that the new file `\user\Alice\letters.txt` has the secrecy label {`AliceInfo, AliceFamily`}. Creating a sub-directory is handled in the same way as creating a file under a parent directory. All files and directories are uniquely identified by their file path and the labels on them do not change after creation.

Deleting a file is also treated as an update to the parent directory and hence the caller must have labels that are the same as the parent directory's. When application requests a directory to be removed, all subdirectories and files will also be removed.

When a file is read, the directories along the file path are also (implicitly) read. However, since labels of files within a directory are at least as restrictive as the directory, Aeolus only needs to check that the process' labels allow the read of the file based on the file's labels. This will imply that the process can also read all the directories along the path. To write a file, a process must have labels that allow the write based on the file's labels. We assume that any exception raised during the write (e.g., exceeding disk quota) does not reveal any sensitive information about the file itself and hence the write does not require the process to also be able to read the file. Table 3.1 summarizes the above label checks and restrictions for basic file operations.

Table 3.1: Information Flow Checks for Basic File Operations

File Operations	Information Flow Constraints
CREATE	Read(Dir)+Write(Dir)
READ	Read(File)
WRITE	Write(File)
DELETE	Read(Dir)+Write(Dir)

Aeolus supports selective declassification and endorsement when writing files, applying authority on the written file rather than the entire process.

- `ReadFile(in F, out buffer)`: Reads file `F` into buffer.
- `WriteFile(in F, in buffer, in listS, in listI)`: Writes content of buffer to file `F`.

The `WriteFile` request has two optional lists of tags, `listS` and `listI`, to specify the tags that are automatically declassified and endorsed as data are written. The file write request is allowed only if the process has authority for the indicated declassification or endorsement. Such selective declassification captures our notion of just-in-time use of privilege.

Another common interface that file systems provide is the file-stream. The file-stream provides a pipe abstraction to the file content. The life cycle of a file-stream starts with the opening of the file-stream, at which time the file is internally bound to the file-stream and the access mode (read-only, write-only, read-write) is defined. Then, depending on the access mode, the user can read (or write) the next group of bytes from the file, pause to process the data, and then continue. When the user is done with the file, the file-stream is closed.

Besides operating with entire files, the Aeolus API also supports file-streams.

- `CreateFilestream(in F, in M, in listS, in listI, out fs)`: Opens a file-stream `fs` for file `F` with access mode `M`.
- `fs.CloseFilestream()`: Closes the file-stream `fs`.
- `fs.Read(in N, out buffer)`: Reads `N` bytes from the file-stream into buffer.
- `fs.Write(in buffer)`: Writes the content of buffer to the file-stream.

File-streams are created using `CreateFilestream(in F, in M, in listS, in listI)` and the access mode `M` is specified as one of `READ-ONLY`, `WRITE-ONLY` or `READWRITE`. For writable file-streams, similar to `WriteFile`, the application can also optionally supply `listS` and `listI`, which are lists of tags that are use for declassification and endorsement of the data being written. Aeolus checks that the process' principal has authority for these tags when the write is performed.

When a file-stream operation is requested, Aeolus checks that the process has the appropriate labels to satisfy the information flow. For a file-stream read operation, the process labels must permit the read based on the file's labels. For a file-stream write operation, the process labels and the file's labels must permit the write (taking into account any selective declassification and endorsement).

Every file-stream read and write is checked to ensure that the process' principal and labels still permit the file operation. While checking at every read and write operations appears to be expensive, we describe techniques in Chapter 5 that are inexpensive.

The rules for file system operations in this section prevent unintended information flow to and from the file system. We ensure that a process cannot leak data to files and process' labels reflect the contamination of files a process reads. In addition to these constraints, we assume standard access control to specify who is authorized to read and write a file using principals in access control lists. Principals are more natural than tags in constraining access. For example, with information flow, a process that can write to a top-level directory can also write to all sub-directories and therefore, using tags in integrity labels alone does not always provide us with the intended constraints (e.g., when files inside a directory should be write-protected from the creator of the directory).

3.8.2 Boxes

Sometimes when an application sends tainted information as a parameter or return value of a call there is a need to control when the callee (or caller, respectively) becomes tainted by that information. For example, the tax-preparer might need to record that it is working for Bob (so that it can send a bill later on) before it looks at Bob's tax information and becomes contaminated with Bob's tag, which it is unable to remove.

One way to solve this problem is to use the file system. By placing the contaminated information in a file and sending the pathname of the file, the caller allows the callee to control when it becomes contaminated. *Boxes* provide this control without requiring the use of persistent storage. A box has outer labels and inner labels, with the constraint that the outer labels must be no more constraining than the inner ones. The contaminated information is inside the box, and the receiver of the box becomes contaminated by it only when it opens the box to obtain its contents. Figure 3-8 shows how the outer and inner labels are used to protect data they contain.

Box labels are immutable. Furthermore, there must be no sharing between the box content and any other objects; otherwise, we could not ensure that the box's inner labels accurately reflect the contamination of its content. Therefore when content is moved into or out of a box, this requires a complete copy.

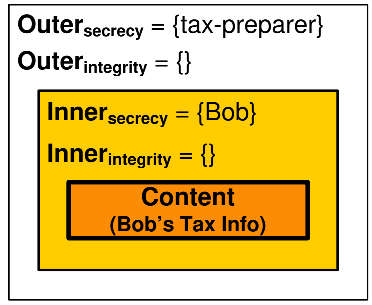

Figure 3-8: Labels on a Box

Box operations:

- `CreateBox(in outerS, in outerI, in innerS, in innerI, out b)`: Creates a new box `b` with the specified labels. The box has an initial value of `NULL`.
- `b.GetInnerS()`: Retrieves the inner secrecy label of box `b`.
- `b.GetInnerI()`: Retrieves the inner integrity label of box `b`.
- `b.GetContents(out content)`: Retrieves the content of box `b`.
- `b.PutContents(in content)`: Copies content into box `b`.

Any process can create a box using `CreateBox(in outerS, in outerI, in innerS, in innerI)`, which returns a new `AeolusBox` object. The outer labels must be no less restrictive than the process' labels. This ensures that the existence of the box cannot be used to convey sensitive information. Aeolus also checks that the outer labels are no more restrictive than the inner ones. With this constraint, our platform can simply check that the process labels allow the read of the box content based only on the inner labels. The callee can use `GetInnerS()` and `GetInnerI()` to find out what labels it needs in order to permit the read or write of the box content. The callee can issue these calls if the process' labels permit the read based on the outer labels. The outer labels are analogous to the labels on a directory and the inner labels are like the labels on a file inside this directory. A process can read the content of a box using `GetContents(out content)`. This is permitted if the process' labels allow the read based on the inner labels of the box. Similarly, a process can write to a box using `PutContents(in content)` but the process labels' must allow the write based on the inner labels of the box.

3.8.3 Shared State

The system as described runs each process in its own private address space and allows no sharing of volatile state; this constraint is necessary in general since otherwise it would not be possible to ensure that process labels accurately reflect the information they use. However, the inability for processes within an app-object to share state is a serious limitation. We address this limitation with *shared state*. Shared state provides a place where shared objects can reside. Additionally, it provides a way for processes to synchronize and exchange messages. Each app-object has its own shared state, and only processes in that app-object can use this shared state.

There are three forms of sharing through the shared state: *shared objects*, *shared queues*, and *shared locks*. `Shared objects` allow the storing and retrieval of objects (e.g., an integer value or an `AeolusBox`) from shared state by different processes. `Shared queues` provide messaging capabilities. `Shared locks` enable synchronization between processes. Like other objects in Aeolus, shared objects, shared queues and shared locks have immutable secrecy and integrity labels.

Shared Objects

Each object in the shared state is identified by a (local) unique ID, refer to as a `SharedObjectID`. `SharedObjectID`s are opaque and are known only to applications that created them. Since processes are isolated, the shared state has a well-known *root object*, named `rootID`, that processes can use to bootstrap communication (e.g., it might store a list of the IDs of other objects in the shared state). The root object has a null secrecy and a null integrity label (i.e., only an uncontaminated process can write to it); its initial value is null.

Processes can create new shared objects, and retrieve and update ones stored previously. In each case, the object is copied completely between the process' heap and the shared state.

Here are the operations on shared objects.

- `CreateObject(in o, out s)`: Creates a new shared object that is a copy of `o` in shared state and returns a new unique `SharedObjectID s`.
- `GetObject(in s, out o)`: Retrieves a copy of shared state object `o` identified by `SharedObjectID s`.
- `ReplaceObject(in s, in o)`: Replaces the current object associated with `SharedObjectID s` with a copy of object `o`.
- `DeleteObject(in s)`: Removes the shared state object associated with `SharedObjectID s`.

Any process can create a new shared object using `CreateObject`. Aeolus returns a new `SharedObjectID` to the application and copies the object into the shared state. The new shared object has the same secrecy and integrity labels as the process. A process can retrieve a shared object provided the specified `SharedObjectID` exists and the process' labels allow the read based on the object's labels. Similarly, a process can overwrite an object using `ReplaceObject` but the process' labels must allow the write. Only processes with null secrecy label can delete a shared object and the process' integrity labels must match the object's; additionally the root object cannot be deleted.

Shared Queues

Shared queues allow users to enqueue objects and to wait for the queue to be non-empty.

Here are the operations on shared queues.

- `CreateQueue(out q)`: Creates a new, empty shared queue object with the process' labels as its secrecy and integrity labels and assigns it a new unique `SharedQueueID q`.
- `Enqueue(in q, in o)`: Appends object o to the end of the shared queue `q`.
- `GetQueue(in q, out o)`: Retrieves the first object in shared queue `q` and removes it from the queue. If the queue is empty, returns null.
- `WaitAndDequeue(in q, out o)`: Blocks until the shared queue `q` is non-empty, returns and removes the first object in shared queue `q`.
- `DeleteQueue(in q)`: Deletes shared queue `q`.

Any process can create a new shared queue using `CreateQueue`. The new shared queue is given the process' labels. Similar to `DeleteObject`, only a process with a null secrecy label can delete a shared queue and the process' integrity label must match the queue's.

When a process appends an object to the queue, the process' labels must allow the write based on the shared queue's labels. All remaining operations (`GetQueue`, `WaitAndDequeue`) read and write the shared queue and hence, the process' labels must be the same as the queue's. If several processes are waiting for a shared queue to become non-empty, the one that has been waiting the longest is awakened when this occurs.

Shared Locks

Although shared queues can be used to implement locks, Aeolus provides a more direct mechanism via `shared locks`.

Here are the operations on shared locks.

- `CreateLock(out k)`: Creates a new shared lock with the process' labels as its secrecy and integrity labels and assigns it a new unique `SharedLockID k`.

- `Lock(in k)`: Attempts to obtain lock on the shared object `k`. If the shared lock is locked, blocks until it is unlocked.

- `Unlock(in k)`: Unlocks the shared lock `k`.

- `DeleteLock(in k)`: Deletes shared lock `k`.

When a process issues a `Lock` operation, the process' labels must permit the read and write based on the shared lock's labels since the process can both observe and influence this shared state. When a process issues an `Unlock`, no acknowledgement of the success or failure of the operation is returned and hence, the process' labels must permit the write based on the shared lock's labels. Similar to `DeleteObject` and `DeleteQueue`, a process must have null secrecy label to delete a shared lock.

3.9 External Communication

Files that the Aeolus Platform has control over are treated as components within our system boundary since we can control the labels that go on these data and prevent the tampering of their labels. Communication to external devices (e.g., I/O devices such as printers) and to nodes outside the system are handled differently. Since we cannot vouch for confidentiality of communication that passes outside the system boundary, the communication is allowed only if the sender's secrecy label is null. Communication from outside the boundary is given a null integrity since we cannot vouch for its validity.

3.10 Authority State

Aeolus maintains *authority state*, which includes information about principals and tags, the principal hierarchy and explicit grants, and authority closures.

3.10.1 Covert Channels

The authority state introduces opportunities for covert channels. This state is modified when privilege is granted or revoked and these modifications can be observed through the use of privilege: a process can determine in this way whether it has been granted privilege for some tag or not. One example is as follows. One process running as principal P_1 creates a set of tags ($T_{leakSet}$) and passes these tag IDs to another process running as principal P_2. These tag IDs are used for encoding a secret protected by tag T. When process with principal P_1 learns about the secret, it becomes tainted by tag T but does not have the authority to declassify. However, it could still communicate this secret to process running with P_2 if it were able to selectively delegate authority for tags in $T_{leakSet}$ to P_2. The process running with principal P_2 can test which of these tags it is authoritative for and leak the secret by writing the result to a file that does not have tag T in its secrecy label.

We avoid these channels by associating a null secrecy label with the authority state and using information flow control. This means that modifications can be done *only* by processes that have null secrecy labels. Thus, the process P_1 cannot perform the delegations after reading the secret. We believe this is a reasonable restriction because modifications to the principal hierarchy are rare and don't typically occur during normal computation. For instance, modifications to the authority state for a medical clinic happen when a new patient joins the system but these changes happen though special administrative actions, not as part of processing the patient records.

There is no integrity label on the authority state since the integrity of this information is guaranteed not by applications but by Aeolus. For example, we allow explicit delegation or revocation only if the requesting process has authority for the tag.

3.10.2 Other Operations on the Principal Hierarchy

There has been a lot of research (e.g., [33, 10]) on the structure of the principal hierarchy and how modifications of the principal hierarchy should be controlled. For example, the notion of ownership is a way to control who can set act-for links. While ideas like these are valuable, they are orthogonal to our work and therefore, we don't go into them. Additionally, we are compatible with richer structures, e.g., ownership and tag and principal deletion. These operations affect modification of the authority state; Aeolus is concerned with using that authority based on the current authority state.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Programming with Aeolus

In this chapter, we present several examples of how the Aeolus programming mechanisms can be used to construct applications. We revisit some of our motivating examples and see how information flow control can be applied using Aeolus.

4.1 Bob and the Tax Preparer

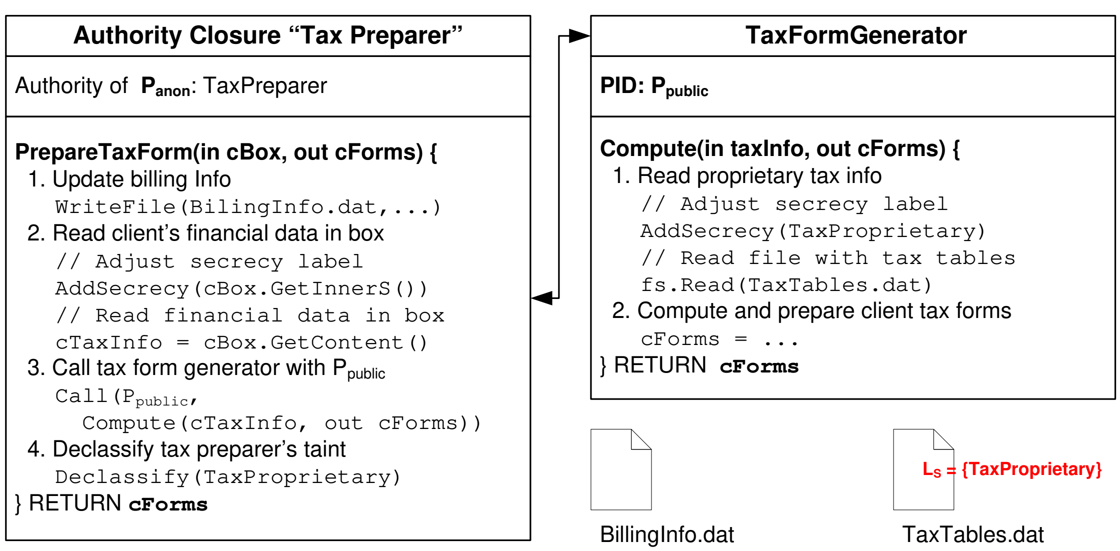

Figure 4-1: Tax Preparer Software Example

Figure 4-1 illustrates the implementation of the tax preparer example; this is implemented as an authority closure, granted authority for the `TaxProprietary` tag.

Bob's confidential financial information is sent in a box `cBox`, so that the tax preparer can record the billing information (in Step 1) before it becomes contaminated. Then, the process secrecy label is augmented so that the process can read Bob's financial data. In Step 3, which performs a complex computation to produce the tax form, can be done with a principal that has no authority and is invoked via a `Call` to run the `Compute` method with P_{public}. Therefore, errors in this code can release neither Bob's nor the tax preparer's confidential information. The `Compute` method raises its secrecy label to read the proprietary tax info and returns the generated tax forms. The caller process becomes contaminated with the `TaxProprietary` tag. In Step 4, the authority closure uses its authority to declassify this tag explicitly; being explicit forces the tax preparer to think carefully about whether it is safe to expose the information. Finally, the code returns the resulting tax forms still contaminated by Bob's tag so that only Bob can use it.

4.2 The Medical Clinic

Figure 4-2 shows a portion of the authority state in a medical clinic. The figure shows the information for an individual patient, `pat`. Patient `pat` has a doctor role `pat-dr` that his doctors act-for; using a role makes it convenient to have more than one doctor and to change doctors over time. The `clinic-admin` also acts-for this role; this link makes it convenient to change doctors, since we can't expect the patient to do this, and a doctor might leave the clinic without making the necessary changes.

Patient `pat-p` has a personal tag, $T_{all-patients.pat}$, which is contained in the label of all his medical records; the doctor role `pat-dr` is granted authority for this tag. This tag is a sub-tag of the $T_{all-patients.*}$ compound tag. $T_{all-patients.*}$ is created by the `clinic-admin` and authority for this tag is delegated to the `billing` closure. In addition, the `billing` closure is also authoritative for the $T_{billing.*}$ tag, which it places in the secrecy and integrity labels of all bills it produces. Placing the tag in the secrecy label prevent unauthorized leaks of the bill, while the tag in the integrity label ensures that the bill was produced by the closure.

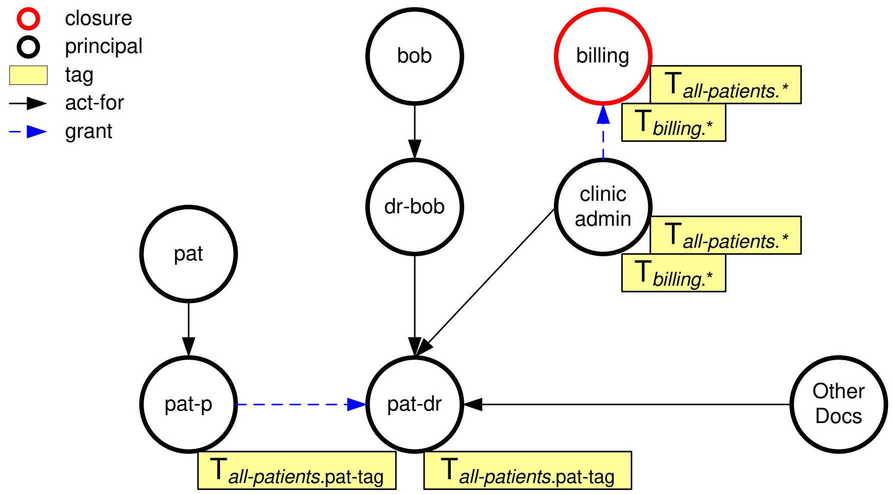

Figure 4-2: Principal Hierarchy for the Medical Clinic Example

4.3 The Sales Analyzer

The sales analyzer is an authority closure that is directly authorized by companies wishing to use its services. Each company maintains a tag (e.g., `SalesA` that appears in the secrecy label of its sales information (e.g., `SalesInfoA.dat`); the sales analyzer is given authority for this tag so that it can produce an unclassified result that the companies can see.

Each time a new company signs up to use the sales analyzer, it needs to supply information about where its sales information resides. When the sales analyzer runs, it needs to find this information as well as the names of the tags delegated to it. This can be accomplished by storing the needed information in a well-known place, e.g., a `Enrollment.dat` file. To ensure that all needed information is supplied when a company signs up, and is removed when it leaves, it is convenient to support the sales analyzer with an `Enrollment Manager` object that provides methods to add and remove companies and to manage what is stored in the file. This file is protected by the `Enroll` tag in its integrity label and only a process running with $P_{StatProvider}$ has authority for this tag. Figure 4-3 shows the tags involved in this example and how they are used to protect the various files. Company A and Company B delegate authority for their sales tag specifically to the anonymous principal of the sales analyzer

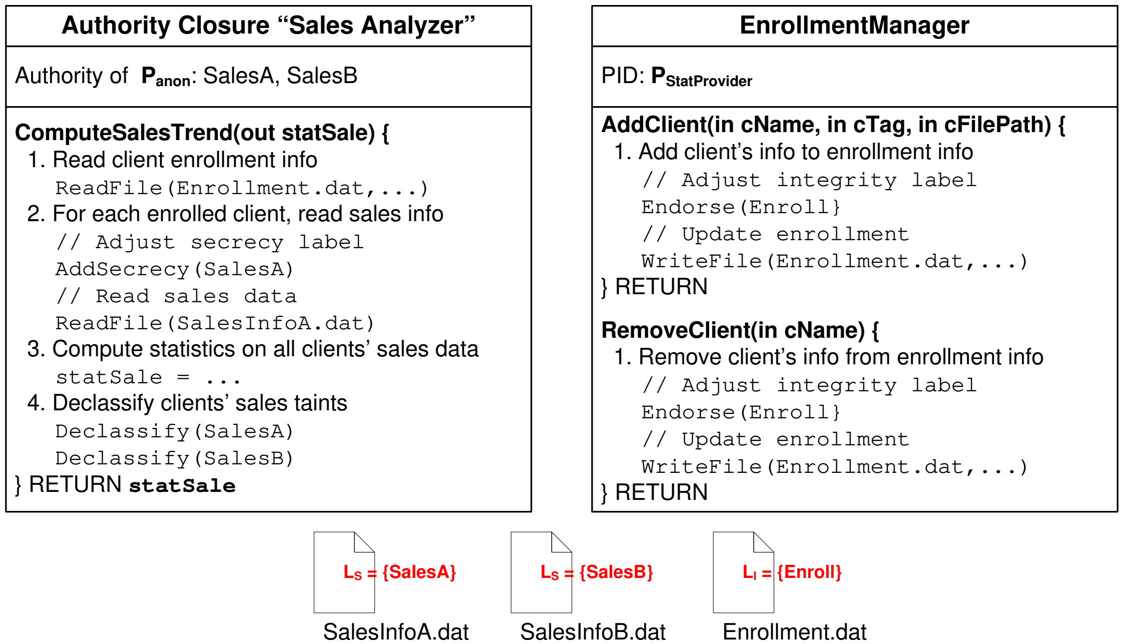

Figure 4-3: Authority Closure in the Sales Analyzer Example

authority closure.

4.4 Job Posting Service

The job posting service is similar to Monster.com. Companies post their jobs but want to prevent certain job seekers, e.g., current employees, from knowing about the posting; meanwhile job seekers submit résumés, but want to prevent their identities from being leaked to certain companies. In this example, we show how a job-seeker can get a list of jobs that exclude the company he/she is working for. The guarantees go both ways, the job-seeker does not know about job postings by his/her employer and the employer does not know that the job-seeker is using this job search service.

It is easy to provide these guarantees using our platform. Each job-seeker has a personal tag (e.g., `alice-seeker`) that tags his or her résumé and this is a sub-tag of the `all-job-seekers` compound tag. Also each company has a tag (e.g., `CompanyAJobs`) that marks its job descriptions. A company provides an authority closure `Match` that is authorized for its tag. This authority closure compares a résumé

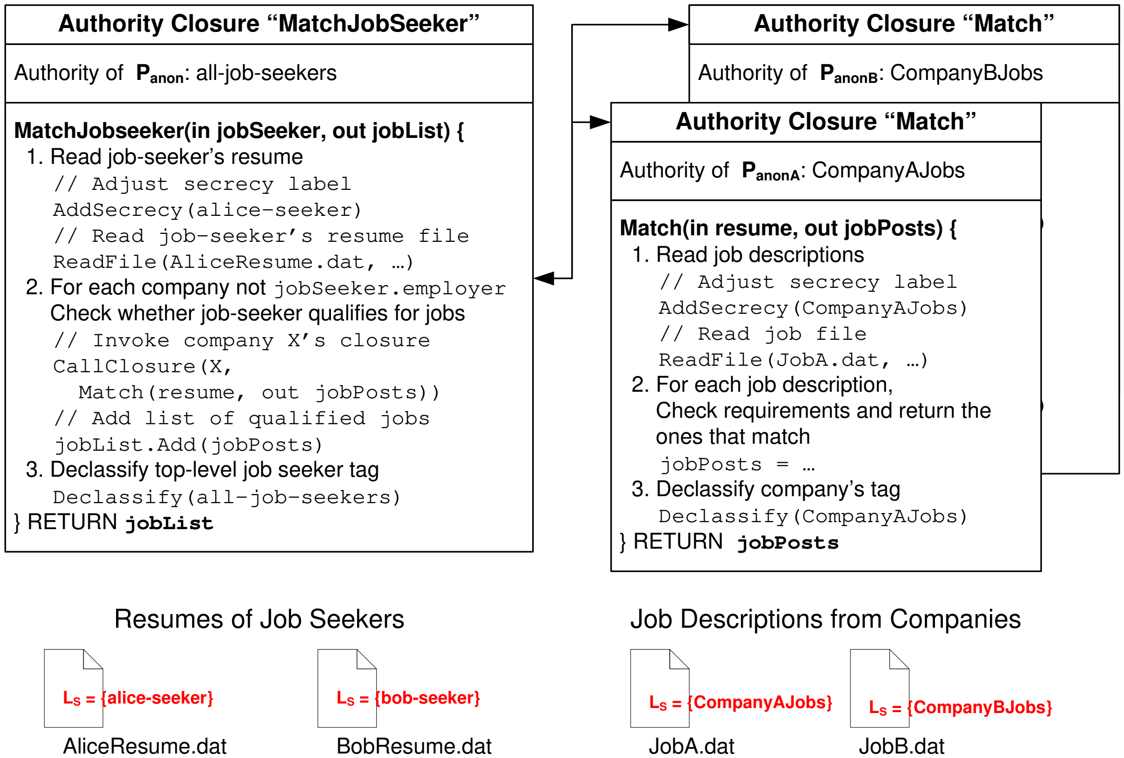

Figure 4-4: Job Posting Service Example

to its job descriptions and returns the descriptions that the seeker qualifies for.

The job posting service provides an authority closure `MatchJobSeeker` that is authorized for the compound `all-job-seekers` tag as shown in Figure 4-4. A job seeker invokes this authority closure to get a list of jobs that he/she qualifies for. This authority closure calls company `Match` closures if the job-seeker does not work for the company and passes them the job-seeker’s résumé as an argument. Each company closure uses its authority for the company tag to declassify the matching descriptions, however, the company cannot leak the résumé since the process is tagged with the job-seeker’s tag. On the other hand, the `MatchJobSeeker` closure can remove this tag and send the result over the network to the poster of the résumé.

4.5 Online Store

The final example concerns a web service that supports online purchases; the example in Figure 4-5 illustrates the use of shared volatile state. A customer engages in a session consisting of a series of interactions during which he examines available items and adds them to his shopping cart; at the end, he either loses interest or proceeds to buy the items. The service has many millions of customers with many thousands of simultaneously active sessions. To handle the load, it uses multiple servers. A call made during a session will return to the same server as the previous call, but if the server is unavailable, some other server can be used.

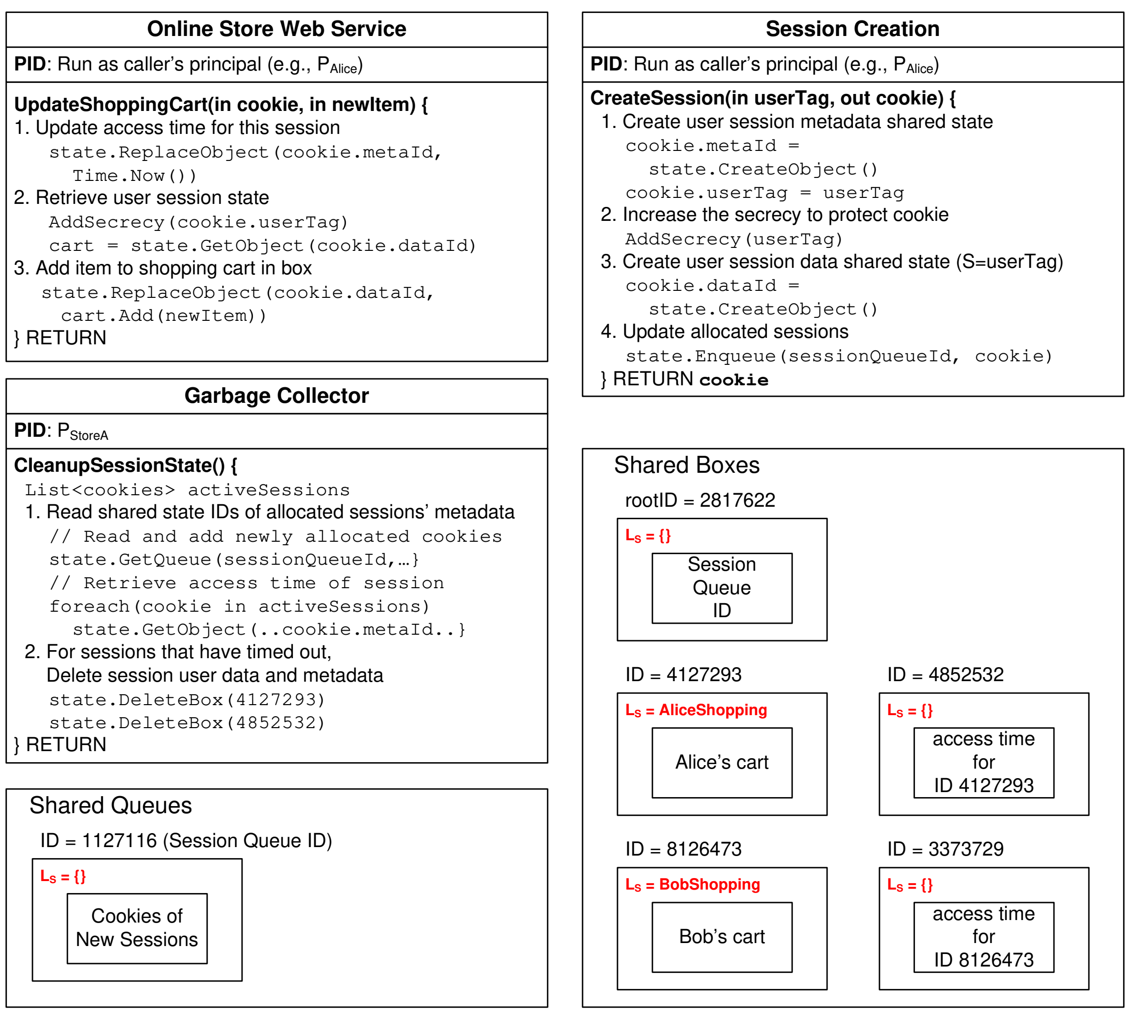

Figure 4-5: Online Store Example

Information about a customer is marked by a unique customer tag (e.g., `AliceShopping`) and a customer's session state is stored at a file server that is part of the system in a file whose secrecy label contains the customer tag. Additionally, the state is stored in the shared state at the server where the request is handled. The server will obtain the customer's session information from its shared state if possible; otherwise it reads the file.

The online store application has a process for each customer request currently being processed, and another process that performs garbage collection on session state. When this application is started at the server, it creates a shared queue and stores its ID in the root object. This queue is used to communicate the creation and termination of sessions to the `Garbage Collector`.

This application requires a way to remove information about abandoned sessions from the volatile state at the servers. This is accomplished by the `Garbage Collector` process; it periodically examines the allocated sessions and deletes information about those that have been inactive for too long. The garbage collector process is notified about new sessions and session termination through the queue.

When a session starts up, the process handling the request creates two objects in shared state. The first contains the session information and has a secrecy label that contains the customer tag. The second is a metadata object; it contains the ID of the session object, and also records the time of the most recent request processed for this session. The metadata object has a null secrecy label. The request handling process then notifies the garbage collector process about the new session by enqueuing an entry containing the ID of the metadata object on the queue. When the request processing is complete, the request handling process stores the ID of the metadata object for the session in the cookie that it returns to the user.

Subsequent requests from the customer such as `UpdateShoppingCart` will include the cookie and the cookie is used to retrieve the customer's session state from our shared state. Processing this request will cause the process to become contaminated by the user tag, but before this happens the request handler process updates the metadata object for the session to reflect the current time.

The garbage collector process maintains a list in which it stores the IDs of the metadata objects for active sessions. It removes sessions from this list when informed about their termination: when it does this, it deletes both the metadata and session objects for that session. In addition, it cycles through the list periodically to identify idle sessions and discards the information for them. The garbage collector is able to do this work without becoming contaminated by the user tags because it never examines the session objects; instead it only looks at the metadata objects.

An alternative structure is to give the garbage collector authority for the user tags; a compound tag could be used for this. With this structure there would be only one shared state object per session, since there is no need to protect the garbage collector from being contaminated. However, the structure described above is safer because the garbage collector process can run without any authority and therefore there is no danger that it will leak information even if it contains an error.

Chapter 5

Distributed Computing Platform

The Aeolus Platform is a distributed computing platform that implements our programming model. Developers can deploy applications on this platform and it guarantees that their computations and data are protected by our information flow control. The platform is designed to support the abstraction and level of control our programming model requires.

5.1 Approach

The *Aeolus Platform* refers to the collection of platform components that run locally and remotely. Figure 5-1 shows the high-level architecture of the platform. Applications run on *compute nodes* and sensitive data are stored on *storage nodes*. Compute nodes enforce information flow guarantees as user applications execute. Storage nodes provide access to file systems that support the Aeolus interface. Compute nodes within our platform consult the shared *authority state* managed by the *authority server* to determine whether privileged operations should be allowed. A compute node and a storage node may co-exist on the same machine or they may be on different machines.

Each compute node runs a set of *platform instances (PIs)*. A platform instance is associated with each app-object running on Aeolus. In addition, there is an *Authority State Client* running on each compute node that manages all interactions with the

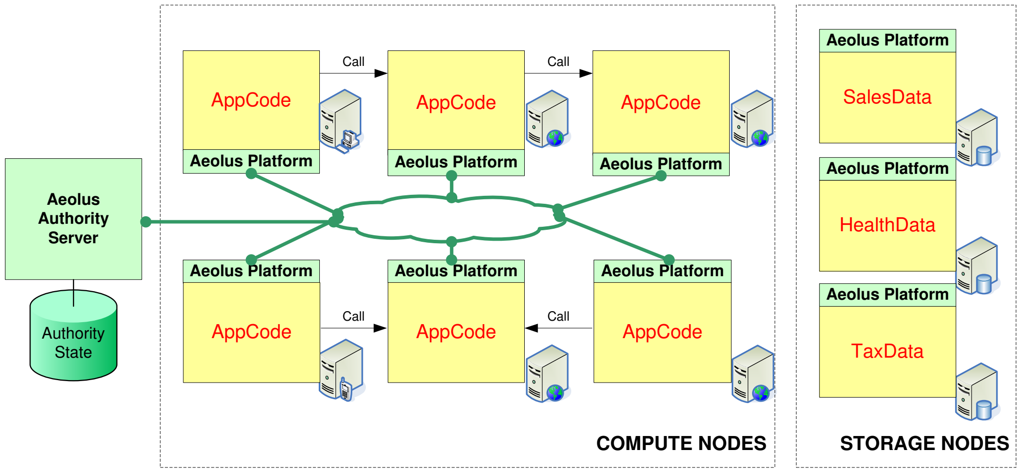

Figure 5-1: High-Level Architecture of Aeolus Distributed Computing Platform

authority server. A platform instance provides a way to sandbox the user application so that the application has no direct accesses to resources (e.g., files, remote calls, I/Os). Instead access to resources must go through our platform, which allows us to ensure that the Aeolus rules are followed.

5.1.1 Isolation

Our platform targets applications that are written in languages such as Java and C#. Programs written in these languages run within a virtual machine such as the Java Virtual Machine (JVM) [61] or the Microsoft .NET Common Language Runtime (CLR) [3]. Our prototype is implemented on top of the .NET Framework but similar design concepts can be applied to Java. We use the isolation properties of these language runtimes to provide sandboxing capabilities in a platform instance. In particular, we use .NET *application domains* (appDomains) to limit the permissions of user code. Furthermore, each application domain has its own address space with no sharing between application domains.

Within each platform instance, as shown in Figure 5-2, there is one Aeolus System appDomain (SYS-d) and many user appDomains (USER-d). User code is executed in a USER-d, which has restricted access, while the SYS-d has full permissions to resources. User applications must use the SYS-d to obtain access to various resources.

The Aeolus SYS-d appDomain provides:

- a launcher that starts up user appDomains
- Aeolus internal state for each user appDomain
- access to shared state
- access to resources such as the file system

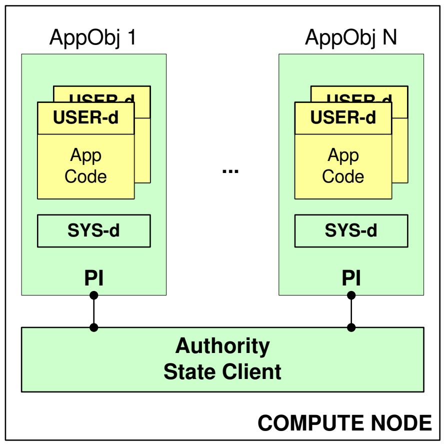

Figure 5-2: Inside a Compute Node

An application domain is an abstraction provided by the language runtime. The language runtime takes advantage of type-safety and memory-safety to allow multiple appDomains to exist within a single OS process. Type-safety ensures, for example, that a program cannot construct an integer value that corresponds to a target address and use it as a pointer to refer to an arbitrary location in memory. Memory-safety guarantees that programs cannot access memory outside of properly allocated objects. Our platform relies on application domains for isolation and therefore, it supports only code that adheres to these language safety properties.

5.1.2 Proxy Object

User appDomains have no access to system resources. Instead each USER-d is provided (when it is launched) with a *proxy object* (PO), which provides methods it can call to interact with Aeolus. The proxy object exposes the Aeolus API (details in Appendix A). It has methods for all Aeolus functions and all external accesses: modifications of the authority state, manipulations of labels, accesses to shared state, file use, I/O devices. In the case of files and I/O devices, the PO provides wrapper code that ensures the information flow rules are obeyed, while using lower level services (e.g., a printer driver to access the real device).

Some calls to the PO involve an inter-appDomain call to the SYS-d, for example, to write to I/O devices since a USER-d has no such permission. Communication from a USER-d to the SYS-d is relatively lightweight. AppDomains communicate with each other via message passing in much the same way as other IPC mechanisms (i.e., marshaling and un-marshaling of values); however, they do not cross OS process boundaries and hence are much cheaper than OS process communication.

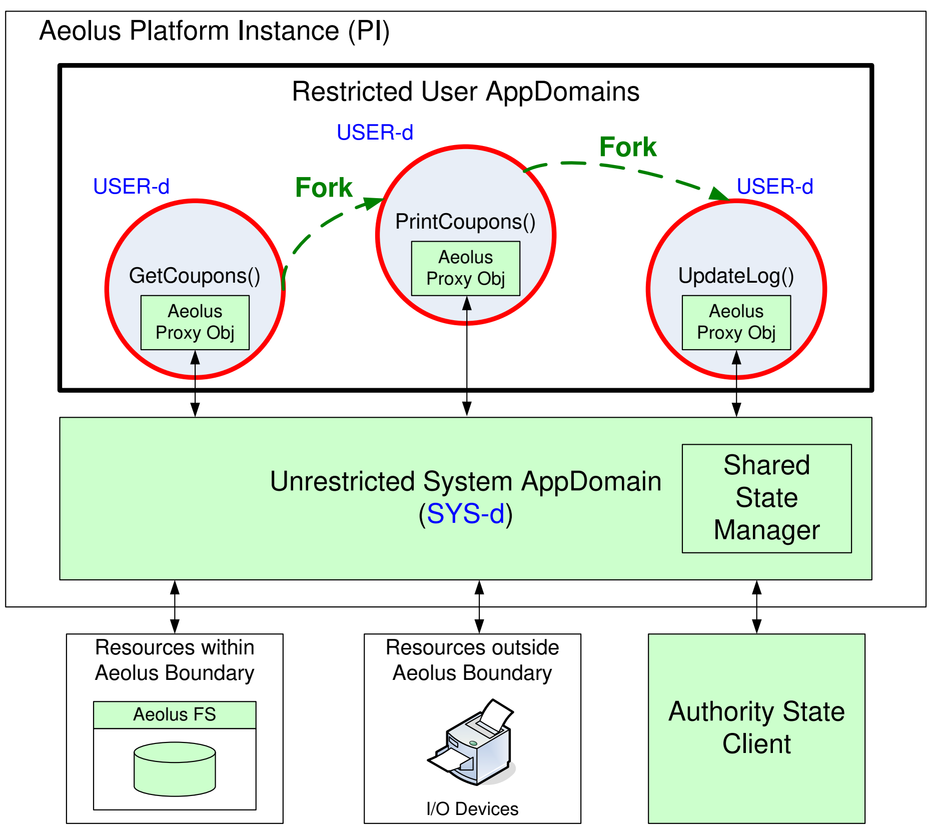

Figure 5-3: Aeolus Platform Instance

Figure 5-3 shows an Aeolus Platform Instance with three user appDomains. Each proxy object stores the process' information flow state (principal ID, process labels) in the USER-d and relies on type-checking and encapsulation to ensure that this state is not misused. The proxy objects can communicate with the SYS-d. The SYS-d can interact with the shared state and the authority state client on the compute node and can access resources both within and outside the Aeolus system boundary.

5.2 Access and Use of Authority State

The proxy object provides methods to accomplish label manipulations and also delegations and revocations. Delegations and revocations require updating the authority state at the authority server. Privileged label manipulations require lookup operations to check whether the process' principal has authority for the tag in question. The SYS-d consults the authority state client for both authority updates and lookups on behalf of the proxy object.

In the case of authority updates, the proxy object checks the information flow state of the process and ensures that the process has a null secrecy label. Then it relays the update request to the SYS-d, which passes it on to the authority state client. The authority state client communicates with the remote authority server to perform the update. Authority lookups may be served locally using information cached by the authority state client and if the needed authority information is not cached, the authority state client retrieves it from the authority server. The authority state client is discussed in Section 5.11.

5.3 Boxes

Boxes allow sensitive data to pass around different parts of an application without increasing the restriction on a process' information flow state unless the content of the box is accessed.

To provide such a functionality, Aeolus needs to control accesses to box content. Our approach is to rely on type-safety by treating boxes as an abstract data type

(`AeolusBox`) with access methods that enforce our information flow rules described previously in Section 3.8.2. Hence, boxes can reside in the user appDomain. Application developers use the proxy object to create a new box, specifying the outer and inner labels of this box. Our platform code generates a new `AeolusBox` object in the user appDomain. Content can be put into a box using the object's `PutContent` method, which checks the process' labels and the box labels before permitting the operation. Similarly, the content can be retrieved using the object's `GetContent` method. These methods do a complete copy by serializing the box contents. This way, we ensure no sharing and therefore, no modifications can be made to box content without going through our checks in these methods. A process can also use `GetInnerS` and `GetInnerI` to retrieve the inner labels of the box and Aeolus checks the process' labels against the outer labels before returning the values.

5.4 Shared State

Shared State is implemented by a *Shared State Manager* that resides within each SYS-d (see Figure 5-3). The shared state manager allocates memory for the shared state and manages accesses to it.

A user application uses the proxy object to create a shared object, queue or lock. This causes an inter-appDomain call to be made from the USER-d of the PO to the shared state manager in the SYS-d. The shared state manager manages the ID space of these objects and returns a new randomly generated ID to the application.

For requests that involve the read and write of a shared object or shared queue entry, a copy of the object to be stored or retrieved is passed across an application domain boundary.

Shared queues are implemented using separate monitors and locks for each queue. Enqueue requests are non-blocking and a copy of the input object is appended to the queue. When the queue is empty, dequeue requests are queued internally. An enqueue request notifies the first requester on this internal wait queue. Dequeue requests are blocking and returns only when notified. Each dequeue request waits in a different thread to avoid blocking the shared state manager.

Similarly, shared locks are implemented using separate monitors for each lock. The shared state manager blocks a caller's request if the shared lock is locked. When it becomes unlocked, the first caller waiting acquires the lock.

5.5 Files

An *Aeolus file system* runs a layer of Aeolus platform code (Aeolus FS) that manages metadata such as file labels to keep track of and protect the storage of sensitive user data.

5.5.1 Aeolus File System

Some previous research projects (e.g., [50, 35]) have tied their design to particular operating system and file system implementations, for example, by using the extended attribute fields of files in Linux-based file systems to encode additional security information. One of our design goals is to be able to support a wide range of file system implementations. Furthermore, we want to focus on understanding how application developers make use of common file system interfaces after the introduction of tags and labels. To that end, our design does not make assumptions about the support available in existing file systems to allow us to annotate files/directories and specify access checks. Instead, we provide a proxy solution that allows us to manage our own metadata while interfacing to different file system implementations.

The Aeolus FS is a TCP server that handles file system operations such as `CreateFile` and `WriteFile`. For each file and directory, the Aeolus FS needs to keep track of its secrecy and integrity labels. These labels are kept as Aeolus metadata and are stored in a relation in a database while the actual file content is stored in the file system. For each directory and file, an entry of the form ⟨`file path, secrecy label, integrity label`⟩ is entered into the database. The `file path` is the full file path from the application user's perspective. Files and directories have immutable labels and these are kept in the `secrecy label` and `integrity label` fields.

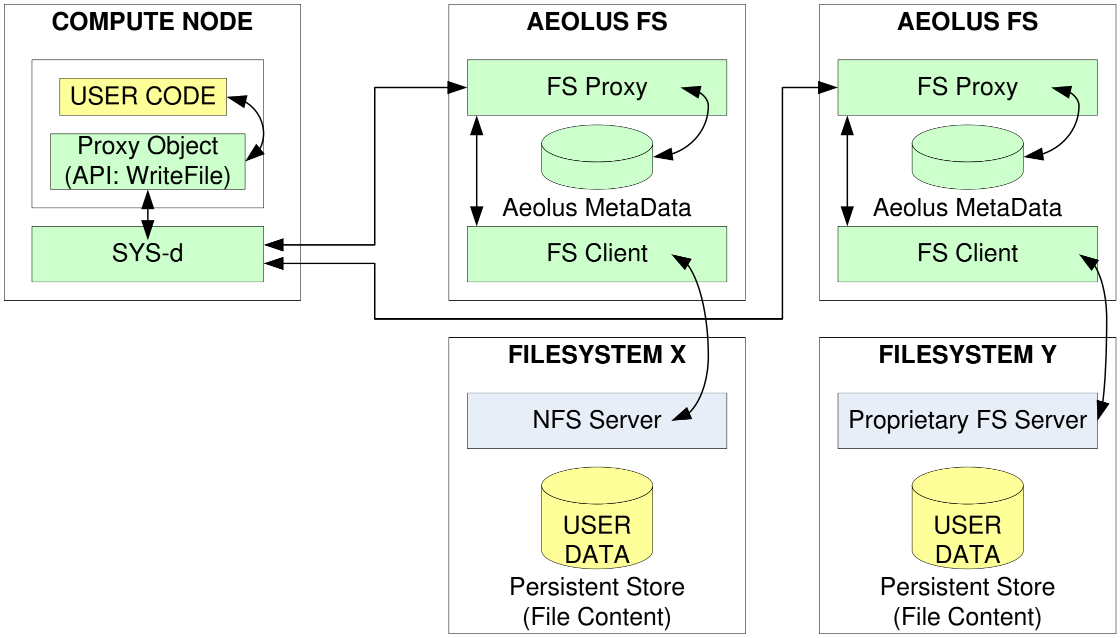

Figure 5-4: Aeolus File System

Figure 5-4 shows how a file operation such as `WriteFile` is handled in Aeolus. The Aeolus proxy object relays the file access request locally to the SYS-d. The SYS-d connects with the appropriate Aeolus FS TCP server responsible for the specified file system (e.g., `/data.csail.mit.edu/` is handled by Aeolus TCP server at port 8400 on the machine with IP address `data.csail.mit.edu`). This request from the SYS-d is annotated with the information flow state (process' labels) of the executing user code. The Aeolus FS TCP server retrieves the relevant label metadata from the database and uses this information flow state to determine whether the file operation should be permitted. If the request passes the label checks, the Aeolus FS updates the actual file content as a client of the underlying file system. Notice that the implementation of the Aeolus FS TCP server is decoupled from the underlying file system storing the user data. The underlying file system is oblivious to the information flow concerns. This allows our system to support file systems ranging from block-based NFS to proprietary file systems.

In our research prototype, we implemented an Aeolus FS for Windows NTFS.

The Aeolus FS TCP server is co-located with the underlying OS file system. This TCP server acts as a FS client to NTFS. It uses the `IO.File` library to perform file system administrative operations and for reading and writing entire files, and uses the `IO.Filestream` library for creating and operating with file-streams.

With the way the information flow checks are set up, most of these file operations that read and write an entire file require only a single check of the process' labels against the file's. Similarly, file maintenance operations such as `Create` and `Delete` require a comparison of the process' labels with those of the parent directory. File labels are cheap to retrieve in our design. The full file path is stored in our metadata database. An index is generated on this attribute, which allows O(1) retrieval of label information based on file path. A traversal of the directory tree is not necessary.

5.5.2 File-streams

Aeolus also provides file-streams, in which the file is read or written in multiple chunks. Application developers use the proxy object to open a file-stream and Aeolus returns an `AeolusFilestream` object. During this operation, the proxy object contacts the SYS-d, which opens the file-stream to the remote file server with the specified access mode and maintains the file handle for this file-stream. The returned `AeolusFilestream` object contains a specific ID in its private fields that the SYS-d can use to refer to this file handle. The application can then issue read and write methods associated with this object.

As discussed in Section 3.8.1, a change to the process state can render an opened file-stream unusable. The straightforward implementation of a file-stream involves checking at each use of the file-stream. This checking involves: the comparison of the process' labels with the file labels, plus in the case of a writeable file-stream, checking against any automatic declassifications and endorsements indicated when the file-stream was opened. However, like Flume [35], we avoid these checks most of the time.

Once a file-stream is opened, Flume performs checks during process label changes and forbids changes that will make the file-stream unusable. We take a different approach and perform checking of file-stream operations. In this way, we associate any such information flow violations with the file-stream operations rather than the process label changes. Thus, the process can change its labels freely provided that at the next file-stream operation, the process labels pass the required check. Such deferred checks can improve performance if the file-stream is no longer used after this state change.

To minimize the label checking required for file-stream reads and writes, we associate a *check* bit with file-streams. When a file-stream is opened, the label checks are done and the check bit is set to false. As long as this bit is false, no label checks are needed for subsequent reads and writes. However, if there is a change in either the process labels or the process authority, the bit is turned on, thus ensuring that the check happens at the next file-stream access so that the appropriate exception can be raised if necessary. If in fact the access is allowed at that point, we turn the bit off. Table 5.1 shows the conditions for setting this bit.

Table 5.1: Conditions for Setting Check bit of a File-stream

Readable File-stream	Writeable File-stream
Declassify	AddSecrecy
Endorse	RemoveIntegrity
Change process' principal	Change process' principal

5.6 Other I/O

A typical I/O device is outside our system boundary. An example of one such device is the native local file system (not running Aeolus FS) that stores files on disks at the compute node. User applications do not have direct access to these file systems; otherwise, sensitive data would be stored to these files and made accessible by any program. A user application runs within a USER-d and is prevented from making these I/O requests.

Access to input devices can be allowed if the process' integrity label is null and access to an output device can be allowed if the process' secrecy is null. This ensures that secrets cannot be leaked and high-integrity process cannot be influenced by low integrity input data.

To support the local file system, we expose the common file system interfaces such as `IO.File` and `IO.Filestream` to the user application in the Aeolus proxy object, which relays the request to the SYS-d. The SYS-d will check that these file operations are performed only when the process has a null secrecy label for writes and a null integrity for reads.

Similarly, support for other devices can be done by exposing the device user interface in the Aeolus proxy object and using the SYS-d to perform label checks prior to issuing the I/O request on the real device.

5.7 Local Forks

Applications running in a USER-d can request a `Fork` on the proxy object. The user application specifies the method to invoke and optionally the PID of the principal, the secrecy and integrity tags that it wants to apply declassification and endorsement to. This method is specified by passing an object (derived from the `AeolusCallable` abstract class) that has a special `Invoke` method (e.g., o.Invoke(), returns nothing).

The `Fork` request is handled in SYS-d. Our platform checks that the current process' labels and PID allow the `Fork` to be made without any information flow violation.

If there are no violations, a new USER-d is set up to run the specified method. Since such setup can be costly, SYS-d maintains a pool of inactive user appDomains, as shown in Figure 5-5. It selects an inactive user appDomain from this pool and sets the information flow state of this USER-d to the specified PID and process' labels. The object with the `Invoke` method is then serialized and copied to this USER-d. (Any arguments for the method invocation are included as part of the object.) Finally, the `Invoke` method starts running in a separate USER-d.

Since user appDomains are reused, our platform must ensure that there is no

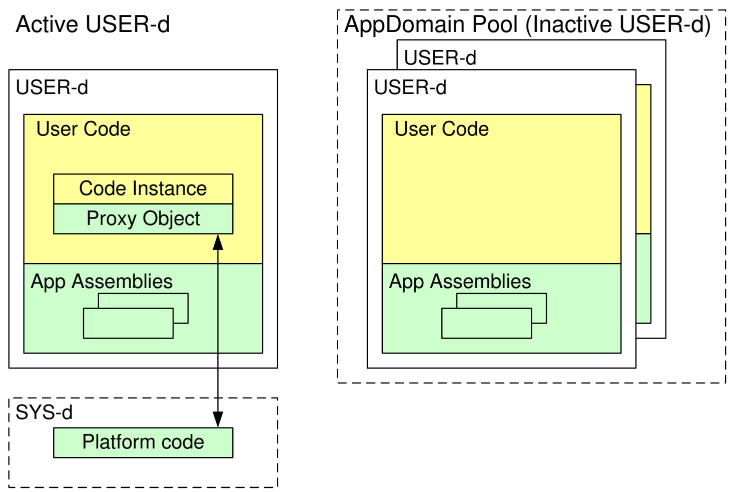

Figure 5-5: User AppDomains

sharing with previous invocations of code in the same USER-d. Hence, in our current prototype, user applications cannot have static variables and our platform checks against this using .NET reflection prior to loading user code into an appDomain.

5.8 Local Calls

Application developers can also run code with different authority by issuing a `Call` on the proxy object. Similar to a `Fork`, the user application passes the method via an `AeolusCallable` object and specifies the PID that it wants to run the method with.

A call runs in the same address space and our platform checks for the same information flow violations as a `Fork`. If there are no violations, the proxy object saves the PID of the caller and issues the `Invoke` method on the object. At the end of the execution, the proxy object restores the PID to that of the caller. Since a call returns to the caller, our system updates the caller process' labels to reflect any additional contamination picked up during the call (i.e., the process label is union-ed with the secrecy label of the caller's and the process label is intersected with the integrity of the caller's).

5.9 Authority Closures

Authority closures are similar to local forks in that they start executing a new method in a different USER-d. However, they run with the anonymous principal ID associated with the closure. The proxy object ensures that the proper code and principal are used.

Recall that an authority closure is identified by a closure ID. When the application makes a closure call, it supplies the closure ID and the method to be invoked via an `AeolusClosureCallable` object. The closure identified by the ID is bound to a key and an anonymous principal and the ID is used for looking up these values. The anonymous principal is used by the platform to identify the authority that has been given to the closure. The key is used to verify the authenticity of the code.

Within each platform instance, the SYS-d has a mapping of closure IDs to ⟨pid, key⟩ pairs. The pid and key values are obtained from the authority state client. After this lookup, our platform must check that the specified code is signed by this key.

All closure code is protected and signed by private-public key pairs. Our platform must verify that the method the application wants to invoke is signed by the proper key. This checking can be computationally expensive if done at the time of the closure call. We minimize this cost by taking advantage of the *strong name* property and reflection in .NET to implement this.

One or more class libraries can be compiled into a *.NET assembly*. This assembly can be signed with a private key at compile time and in doing so, a strong name is associated with it. A strong name consists of the assembly's text name, a public key, a version number and optional build information. To get a valid strong name, an assembly is strong-name signed during the build process; this is done by using RSA to encrypt the SHA1 hash of the assembly [17] with the public-private key pair of the code publisher. We use this public key in the strong name as the key for Aeolus authority closures. Therefore, all authority closure code must be packaged into assemblies with strong names.

When a strong-name assembly is loaded, .NET computes the hash of the assembly and checks it against the certificate signed by the code publisher. Therefore, our platform can safely refer to the public key of the strong name to confirm that the code is authenticated by the expected key. Figure 5-6 shows the structure of a strong-name signed assembly that contains the authority closure code for the code object and how its strong name and certificate are used for the checking.

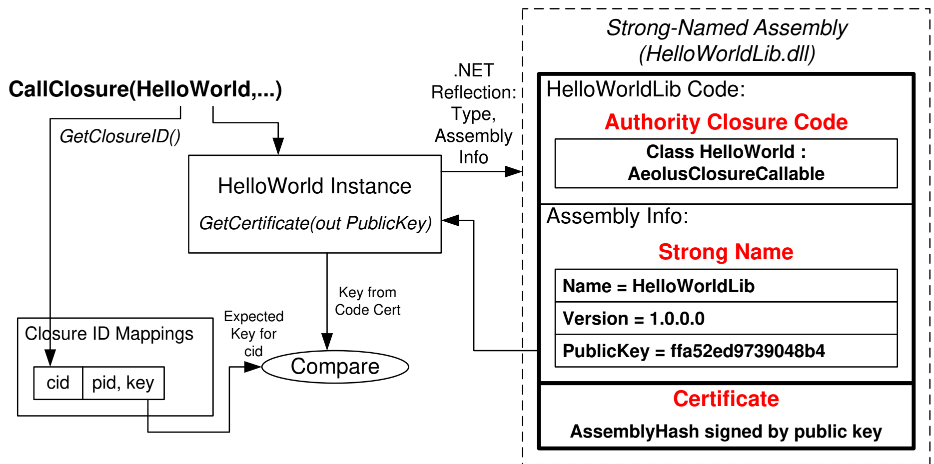

Figure 5-6: Using Strong Name to Verify Authority Closure Code

The proxy object looks up the strong name of the supplied `AeolusClosureCallable` code object using .NET reflection. It checks that the public key in the strong name is the same as the key from the closure ID mapping. After this check, it starts the `Invoke` method in a new USER-d but with process' principal set to the anonymous principal from the closure ID mapping. Similar to fork and call, the caller can also specify the list of tags to apply declassification and endorsement to the labels for the closure. The proxy object checks that the caller principal has authority for these tags. At the end of the closure call, the result object is returned and caller's labels are updated to reflect any additional contamination as a result of the closure call.

5.10 Remote Procedure Calls

Aeolus supports one form of remote procedure calls called *web methods*. Like other remote procedure call mechanisms, web methods allow a client to call a method that is on another node. In our prototype, we allow user applications to invoke web methods only on compute nodes that are within our system. Our platform ensures that the web method is executed with the appropriate principal and process labels to preserve information flow properties. Hence, we need to communicate information flow state and mediate the web method call between the client and server compute nodes. Our approach is to tap into the messaging framework and web method runtime environment and take advantage of their programming hooks and callbacks.

Web methods are exposed by application servers using a common description language called the *Web Service Description Language (WSDL)* [42], established by the World Wide Web Consortium. Web methods are invoked over HTTP and are hosted by an HTTP server such as the Microsoft Internet Information Services (IIS). They use the Simple Object Access Protocol (SOAP) to encode call arguments and return values. SOAP is a XML-based protocol that lets applications exchange information over the Internet. At the remote machine, the HTTP server passes the SOAP message to the language runtime, which de-serializes the message and invokes the web method. Aeolus taps into the programming hooks in the serialization and de-serialization of SOAP messages at the client and server to inject and retrieve information flow state for the call and return. Our platform code also taps into the dispatching logic for web method invocations on the server-side to control their invocations. Next, we present a step-by-step walk-through of a web method invocation on our platform.

Applications include references to the web service (set of web methods) they want to invoke. This reference is a URL pointing to the web service interface. From the application's perspective, a web service is a remote object that contains a set of web methods. (However, these web methods are often independent, stateless, and atomic.)

Figure 5-7 shows how a web method invocation is handled in our system. On the client side, to a user application, invoking an Aeolus web method seems no different from invoking a normal web method.

In Step 1, a remote call is routed to the .NET web service runtime. The .NET web service runtime handles the serialization and de-serialization of messages to/from the web service and Aeolus registers several callback functions on these message events.

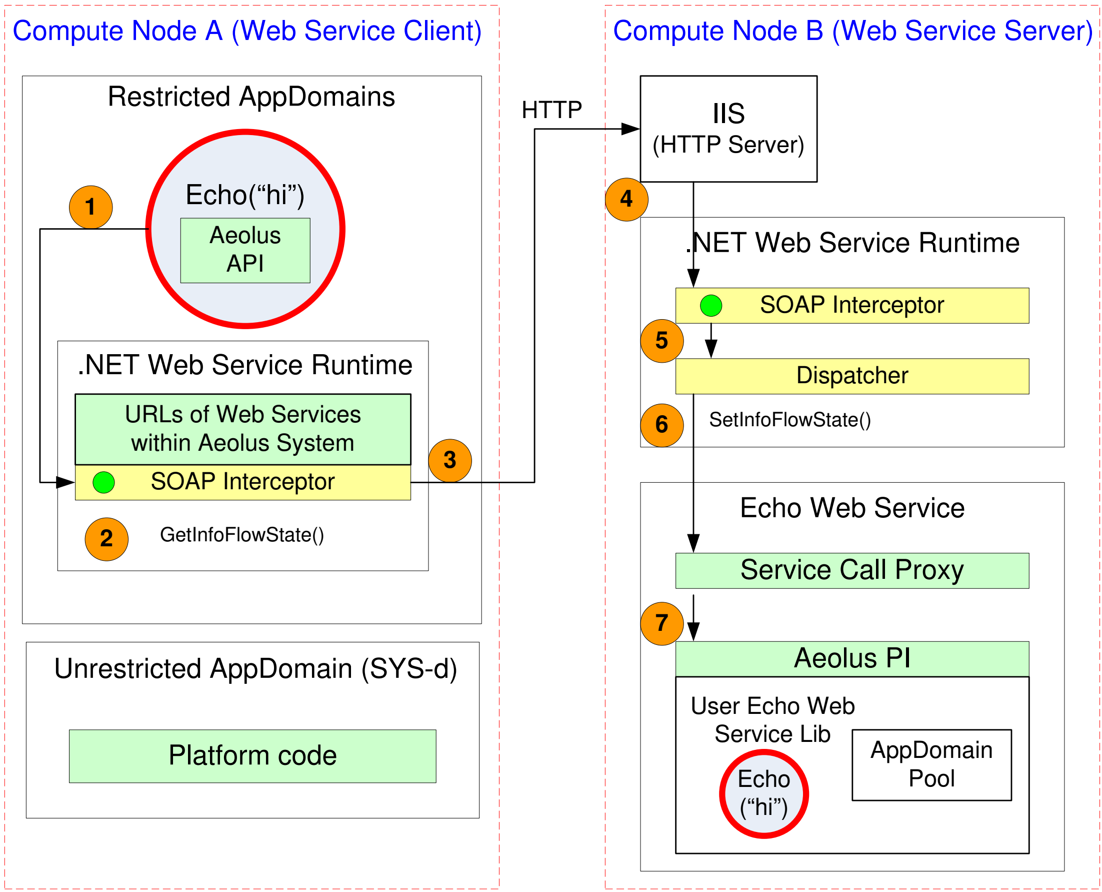

Figure 5-7: Web Method Invocation on Aeolus Platform

When Aeolus receives an outgoing message, it checks that the message is destined for a node within our system. Aeolus does this by consulting a list in SYS-d that includes the IP address and public key associated with each trusted node within our system. This information is part of the authority state and the public key is the basis for creating an SSL connection between nodes within our system. Then, the remote method call is serialized into a SOAP message and Aeolus attaches an `Information Flow State Header` containing the principal ID, and secrecy and integrity labels of the caller to the message in Step 2. This extended message is sent over HTTP to the remote web service server in Step 3.

The HTTP server at the remote machine processes the incoming network request and directs it to the server-side .NET web service runtime in Step 4. This runtime is responsible for dispatching requests to the web service implementation. A web service implementation that runs on the Aeolus platform must wrap the user implementation with our platform code to control how the user web method is executed. In our current prototype, this is done manually by taking in the user implementation as a class library and creating a `Service Call Proxy` that has call stubs for the user methods. Each call stub wraps the user method with platform setup and teardown code. The `Service Call Proxy` is deployed as a web service on the server. When this web service runtime receives the SOAP message at the server in Step 5, the SOAP interceptor first checks that the message comes from a node within our system. Then it extracts the `Information Flow State Header` attached to the message. In Step 6, the web service dispatcher relays the call to the Aeolus `Service Call Proxy` with the needed information flow state to execute the web method. Our platform code selects an unused USER-d from a pool of user appDomains and initializes it with the caller's information flow state. The user web method is invoked and when it finishes execution, the process' information flow state is extracted and sent back to the client in an `Information Flow State Header` along with the message containing the call return value. The SOAP interceptor on the client retrieves this header and updates the caller process' information flow state accordingly.

In our prototype, we limited remote calls to nodes within our system. However, this can be extended to handle calls from nodes outside by running the remote method with P_{public}. If the caller is outside our system, the code must run with null integrity label and can only return to the caller if it has a null secrecy label. If the callee is outside our system, Aeolus will only allow the call to be made if the caller process has a null secrecy label and upon its return, the callee will have a null integrity label.

5.11 Authority Management

So far, we have discussed how our platform can monitor data channels and track information flow. Platform instances use authority state to determine whether privileged operations should be allowed. Management of authority state is an important part of the system.

We expect lookups of authority information to be frequent whereas updates occur relatively rarely. Lookups are part of the normal execution of an application and happen as data are declassified or endorsed. Updates occur primarily as new principals are added or removed, for example, when a doctor leaves the clinic, and during re-assignment of responsibility, for example, when a patient is assigned a different doctor.

The Aeolus platform is intended to support many compute nodes from different organizations/institutions, hosted on different and geographically dispersed data centers. With roughly 10,000 nodes per data center and many applications running simultaneously on each node, this can result in millions of authority lookup requests per second. A design in which authority is checked by a centralized authority server is infeasible with this high aggregate number of authority lookups. Not only does this central authority server need to serve many compute nodes, but the computation needed to check authority can be time-consuming, requiring the traversal of the principal hierarchy and per-principal authority state to determine whether principal P indeed has authority for tag T.

Also, authority lookups can be in the critical path of application execution. For example, prior to writing data to a file, the application may need to make its secrecy label less restrictive by calling `Declassify`. The application cannot proceed until the process' secrecy label has been updated. To do so, the Aeolus PI needs to check that the running principal has authority for the specified tag.

Our approach is to handle authority updates and lookups separately. Our architecture consists of *authority server(s)* that handle updates and *authority state clients* on compute nodes that maintain cached state to serve lookups. Most lookups are handled locally. This approach reduces latency in looking up authority information and reduces load on the AS.

Authority state is stored persistently at the AS. In our prototype, we use one server, however, we can replicate the authority state on a group of servers to provide high reliability and availability. Furthermore, we can partition the authority state among many ASes. Each authority server could be responsible for one independent part of the authority state to reduce the chance of conflicting updates. An authority server could also be kept close to where it might be likely used. We plan to explore these approaches as future work.

5.11.1 Structure of the Authority State

This section describes how we store authority state at the authority server (AS).

In Aeolus, authority state includes information about tags and principals, the principal hierarchy, delegations that have been granted to different principals, authority closures, and system membership. This authority state is needed to determine not only whether a principal has authority for a tag but also to find actors and actees of principals. The authority state must have sufficient information to capture these decisions.

The authority state is stored in several tables. Table 5.2 shows the attributes maintained by each of these data tables.

Table 5.2: Authority State Data Tables

Principal Table `Pid`: Principal ID
Tag Table `Tid`: Tag ID
ActFor Table `PidFrom`: Principal ID of Actor `PidTo`: Principal ID of Actee
Grant Table `PidFrom`: Principal ID of Grantor `PidTo`: Principal ID of Grantee `Tag`: Tag granted
Closure Table `Cid`: Closure ID `Key`: Public key used to verify closure code `Pid`: Anonymous Principal ID
Membership Table `Node`: Node identifier `IP address`: Assigned static IP address of node `Public Key`: Public key

The `Principal Table` and `Tag Table` are used to identify valid tags and principals in the system and the principals that created them. The `ActFor Table` encodes the principal hierarchy by listing all the edges of this directed acyclic graph connecting actor principal (`PidFrom`) to actee principal (`PidTo`). The `Grant Table` encodes delegations by listing all edges in the delegation chain for specific tags.

The `ActFor Table` represents implicit authority whereas the `Grant Table` represents explicit authority. To determine whether a principal P has authority for a tag T, one uses the `ActFor Table` to traverse the links, starting with P in `PidFrom` and retrieving all actees of principal P. For each of these actees, one can then use the `Grant Table` to compute the tags that this actee is directly authoritative for and check whether T is in this set. The set of explicit authority for a principal consists of all the tags in which the principal is the grantee in the `Grant Table`.

When a principal creates another principal, the new principal ID is recorded in the `Principal Table` and since the creator principal automatically acts-for the new principal, a link is added in the `ActFor Table`. When a principal creates a tag, the new tag ID is recorded in the `Tag Table` and the creator principal is automatically granted authority for the tag. This is done by creating a new entry granting the creator principal the authority for this tag (with the system principal P_{root} as the grantor principal) in the `Grant Table`.

Revocation causes entries to be removed from these tables. When a delegation is revoked, this will affect delegations made transitively. For example, if principal P_1 granted tag T to principal P_2 and then P_2 grants this to principal P_3, there will be two entries in the `Grant Table`: $\{P_1, P_2, T\}$ and $\{P_2, P_3, T\}$. When P_1 decides to revoke the delegation of tag T to P_2, the first entry is removed. We also remove the second entry if P_2 has no other delegations for tag T. In this way, the `Grant Table` is kept up-to-date; we can retrieve a principal's explicit authority by simply reading this table and do not have to check whether the delegation is still valid.

When an act-for link is revoked, we remove the corresponding entry in the `ActFor Table`. Since implicit authority is computed dynamically, other links in this table are not affected.

Our system provides random IDs for principals, tags and authority closures. Tags are encoded as a pair of IDs: $\langle$ID1, ID2$\rangle$. Top-level tags have a unique ID for ID1 and a 0 for ID2. Sub-tags have ID1 of the compound tag they belong to and a non-zero unique ID2. This way, we are able to represent groups of tags without having to maintain explicit data structures for them.

The `Membership Table` maintains the list of machines that are trusted to be running Aeolus. The `Node` attribute represents the globally unique identifier of the machine. The entry also includes the static `IP address` assigned to this node and its `public key`.

5.11.2 Caching

We maintain cached authority state at authority state clients on compute nodes. Use of cached state offloads the AS and speeds up the processing of operations. However, we need to partition the authority state effectively for reasonable cache performance.

Partitioning Authority State

At one extreme, authority state clients can obtain and cache the entire authority state. In this way, they have a complete copy. However, authority state of the entire system can be large since it includes information on all principals and tags in the system, across many organizational departments and institutions. We want Aeolus to be able to run on compute nodes with a wide range of hardware resources including mobile devices, which are becoming popular entry points for invoking distributed computations [25]. The authority state of the system may not fit on devices with small memory capacity. Moreover, transferring the entire state can be time-consuming.

We need to partition the state effectively to get a reasonable tradeoff between minimizing the number of fetches required over time and the size of each fetch. The partitioning requirement is the usual one: a fetch should bring over related information that is likely to be useful in the future.

Our solution is to divide the authority state into *cores*. Each time a client requests some information from the AS, it receives a core. Additionally, we give the application a way to control which core information goes into. When a principal is created, the caller can indicate whether it should go into a new core or an existing one. In the latter case, the caller can specify the core by providing a PID as an extra argument; the new principal will be placed in that principal's core, or in the core of the caller's principal if the argument is missing. Tags and closures always go into some principal's core; again there is an optional argument to specify the principal and if the argument is omitted, the tag or closure goes into the core of the creator's principal.

A core stores not only all principals, tags, and closures that have been placed in it, but also all the acts-for and delegations for these principals. In addition, it stores sufficient information to follow acts-for links to principals that reside in other cores. Figure 5-8 shows the clinic example discussed before and how the principals and tags are grouped into cores. The figure shows a patient core and a doctor core for the doctor `dr-bob` who acts-for the patient's doctor role `pat-dr`. The figure also shows the clinic core. This core would have been created when the clinic system started running and contains various entities related to managing the clinic, including (for this example) the `clinic-admin` principal, the `billing` closure, and the `all-patients` compound tag. The figure omits a link between the `all-patients` tag and its subtag, `pat-tag`) since this information is not stored explicitly but instead is captured through tag names.

This partitioned authority state is implemented using the same tables as Table 5.2 but each entry has an additional attribute that identifies the core that a table entry belongs to. A core consists of all entries from all data tables that have the same `CoreId` and is used as the unit for a fetch. The `ActFor Table` and `Grant Table` contain additional attributes (shown in bold font in Table 5.3) on how to find the cores of the principal that they are linked or chained to. The `Membership Table` is not partitioned since its information is made available to all authority state clients.

An authority state client fetches an entire core from the AS each time. When a core is fetched, the authority state client receives all table entries with the requested core ID in `CoreId` and stores them locally in in-memory tables structured like the

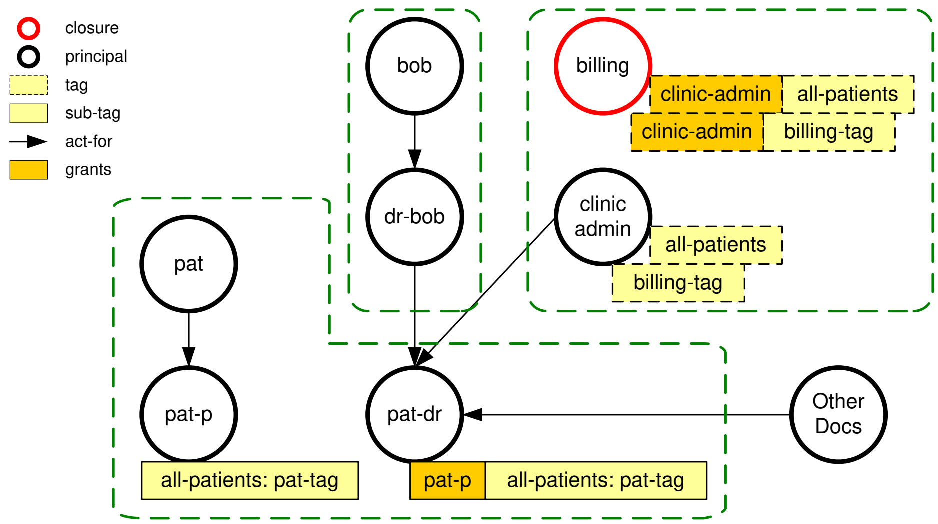

Figure 5-8: Structure of Core for Medical Clinic Example

Table 5.3: Partitioning Authority State Data Tables by Cores

Principal Table `CoreId`: **Core ID** `Pid`: Principal ID
Tag Table `CoreId`: **Core ID** `Tid`: Tag ID
ActFor Table `CoreId`: **Core ID (core of Actor)** `CoreTo`: **Core ID (core of Actee)** `PidFrom`: Principal ID of Actor `PidTo`: Principal ID of Actee
Grant Table `CoreId`: **Core ID (core of Grantee)** `CoreFrom`: **Core ID (core of Grantor)** `PidFrom`: Principal ID of Grantor `PidTo`: Principal ID of Grantee `Tag`: Tag granted
Closure Table `CoreId`: **Core ID** `Cid`: Closure ID `Key`: Public key used to verify closure code `Pid`: Anonymous Principal ID

ones at the AS.

When an user application performs a privileged operation that asks whether principal P has authority for tag T, this may require fetching multiple cores. Figure 5-9 shows a sample traversal that spans two cores. Assuming that the authority state client at the compute node has neither core cached, when it is asked the question `HasAuthority(`P`, `T`)`, it first checks whether it has information on principal P by consulting its cached `Principal Table`. If P is not found and since it doesn't have any information about this principal (i.e., it doesn't know which core P belongs to), it issues a special request to the authority server to `FetchCoreForPrincipal(`P`)`. The authority server has the complete authority state and can look up the core of principal P and in this example, it will return `Core` C_1. Then, the authority lookup continues exploring the principals that P acts-for by using the local `ActFor Table`. When it finds an actee principal, it checks the `CoreTo` attribute to see if the required core is cached, if not, it issues a request to `FetchCore(c)`, which will return `Core` C_2 in this example. With this information, it can check whether its actees have the authority for T by looking at the local `Grant Table`.

Notice that `FetchCore` is preferred over `FetchCoreForPrincipal` whenever possible, as it alleviates the extra computation at the AS to find the core that a principal belongs to.

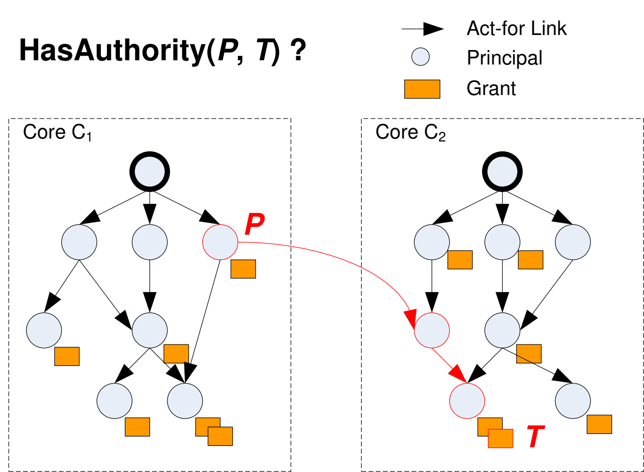

Figure 5-9: Traversal across Different Cores during an Authority Lookup

Structure of the Client Cache

Aeolus platform instances use the local authority state client to reduce the latency of authority lookups. When cores are fetched from the AS, they are stored in a *core cache*. The core cache is a set of in-memory tables storing a partial authority state of the system.

Even when the needed cores are in the core cache, however, questions about whether a principal has authority for a tag can be costly to answer because they require finding one path among many. For example, a `doctor` acts-for many patients, and only one of these paths provides authority for a particular `patient-tag`. The local authority state client has a two-tier caching structure. When an authority lookup is requested, it first checks in a fast *content cache* and if the request cannot be answered, it consults a larger and slower *core cache*.

The content cache allows frequently asked questions, which may involve longer paths, to be answered quickly. It is likely that if a particular question about authority is answered now, it will be asked again in the near future. The content cache stores recently computed answers. This cache stores pairs $\langle principal, tag \rangle$; each entry indicates that the principal has authority for the tag. It is implemented as a hashtable; the hash of the pair serves as the key. The content cache is consulted first in an authority lookup. In this way, we can take advantage of temporal locality of lookups within the same application and within user sessions. Upon a miss in the content cache, the core cache is queried and the answer is added to the content cache.

Cache Management

The content cache and core cache are not unbounded in size and are limited by the physical memory constraints on compute nodes. When these caches fill up, an eviction scheme is invoked to select victims. Ideally, victims should be selected such that they are not likely to be used again. For both the content cache and the core cache, we use the least-recently-used (LRU) scheme [16] as it is likely that entries used in the recent past will be used again in the near future.

Both caches use a doubly-linked list to maintain access order of cache entries. The head of the list refers to the least recently used entry where an eviction victim is extracted from. When an entry is accessed, it is moved to the end of the list. For the content cache, an entry is moved to the back of the list when an authority lookup uses the entry, which provides the desired recently used behavior. However, for the core cache, the list is updated only in the case of a miss in the content cache and hence a core that has been used recently via the content cache might be discarded from the core cache.

5.11.3 Synchronization and Update

Authority updates run at the authority server, while authority lookups are served locally by an authority state client with occasional communication to the AS to fetch specific cores.

We store the authority state in a database at the authority server. This is convenient since it allows us to take advantage of database transactions to ensure serializability and atomicity for performing updates that can affect several tables and many cores and for handling concurrent client requests. Additionally, the database allows us to run queries to determine the current system state; since queries are serialized relative to updates, we can be sure the results are consistent.

An authority update can modify a number of cores while an authority lookup fetches specific cores. If we are not careful, this can lead to an inconsistent state in the local caches. For example, suppose authority update U modifies both core C_1 and core C_2, and suppose also that an authority state client has a copy of C_1 in its cache. If the authority state client makes a fetch request for core C_2, its cache might end up in a state where its C_1 does not reflect the effects of U, while its C_2 does.

It is important to present a consistent view of authority state to applications. For example, the authority state may contain a portion of a principal hierarchy that has a conflict-of-interest constraint. Bob can work on behalf of Company A or Company B but not both. Bob used to work for Company A and this allowed Bob to act-for `CompanyAEmployee`, which is recorded in core C_1. Later, he leaves to join Company

B. Company A revokes Bob's ability to act-for `CompanyAEmployee` and Company B recognizes Bob as its new employee and allows Bob to act-for `CompanyBEmployee`, which is recorded in core C_2. The conflict-of-interest constraint will be breached if an application sees core C_1 prior to the updates and core C_2 after the updates.

Our platform avoids this problem while still allowing fetches of individual cores. Our solution is to take periodic *snapshots* of the authority state and have servers fetch requests from them. Each snapshot provides a consistent view of the database. Additionally, we ensure that each authority state client's cache is consistent with the most recent snapshot. This way, we can avoid having fetches cause inconsistencies. For the example above: if the snapshot was taken after the update U, then it reflects changes to both C_1 and C_2; otherwise it reflects neither change. And so long as all cores in the authority state client's cache are from the same snapshot, they will be consistent. For example, the authority state client will either see a state before U or after U.

Clearly, this scheme allows authority decisions to be made based on old information. We assume applications can tolerate using a slightly out-of-date version of the authority state. For example, changes to a hospital's employee list only need to be propagated daily. However, the system guarantees that this information isn't very old: snapshots are taken frequently, based on system parameter, Δ. For example, Δ might be set to 30 seconds, which ensures that all authority decisions are based on information no more than 30 seconds old.

To make this scheme work, we need an efficient way to produce snapshots and an efficient way to keep server caches up to date with the most recent snapshot.

Producing Snapshots

In our system, the authority server maintains the most recent version of the authority state as shown in Figure 5-10. The authority server serializes authority update requests from different clients and manages a single write-copy of the authority state. Examples of authority updates include `CreatePrincipal`, `ActFor`, and `Delegate`. As discussed earlier, this server uses a database to store the authority state over several relations. In addition, it maintains an *operation log* (`Op Log`) that contains ordered information about successful authority updates. The operation log allows our system to keep track of the order of updates with respect to when various snapshots are taken.

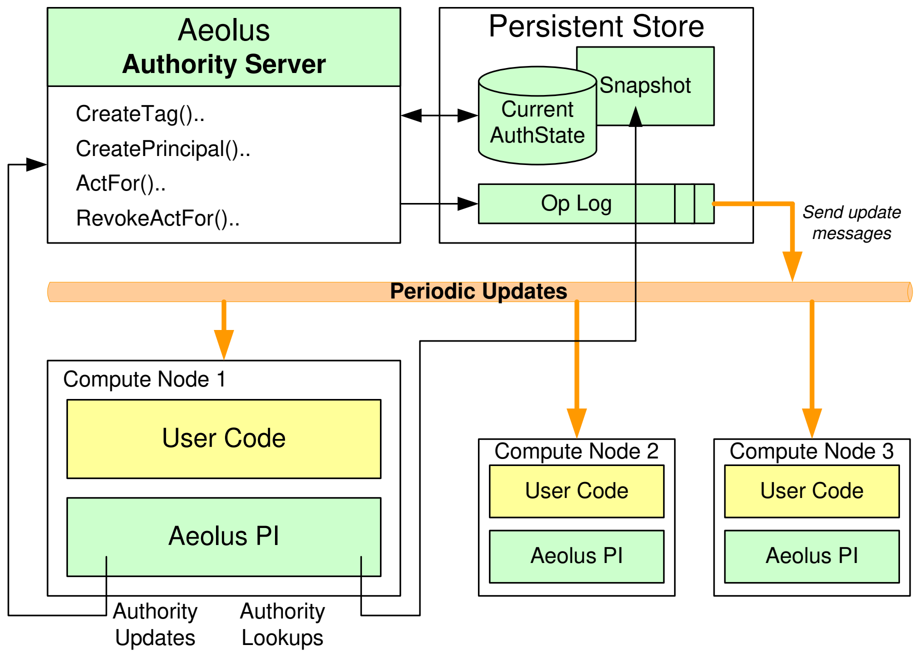

Figure 5-10: Authority Server

An update request may change several relations and cores and it is executed as a database transaction. If the transaction commits successfully, the authority server assigns the update operation a sequence number and appends an entry to the operation log. The entry includes the type and input parameters of the authority update operation as well as the results of applying this operation on the current authority state (e.g., the newly generated tag ID for `CreateTag` operation). In this way, one can parse the operation log as a redo log to construct mirror versions of the authority state.

The authority server is also responsible for creating snapshots of the authority state periodically. It does this by making a request to the database to create a low-overhead copy-on-write snapshot of the authority state. A new database snapshot is generated and an unique snapshot name is returned to identify this snapshot. The authority server adds an entry to the operation log to record this snapshot creation.

Fetches are directed at the most recent snapshot broadcast in update messages, as shown in Figure 5-11. The authority state client must be informed of the name of this snapshot so that it can fetch cores from it.

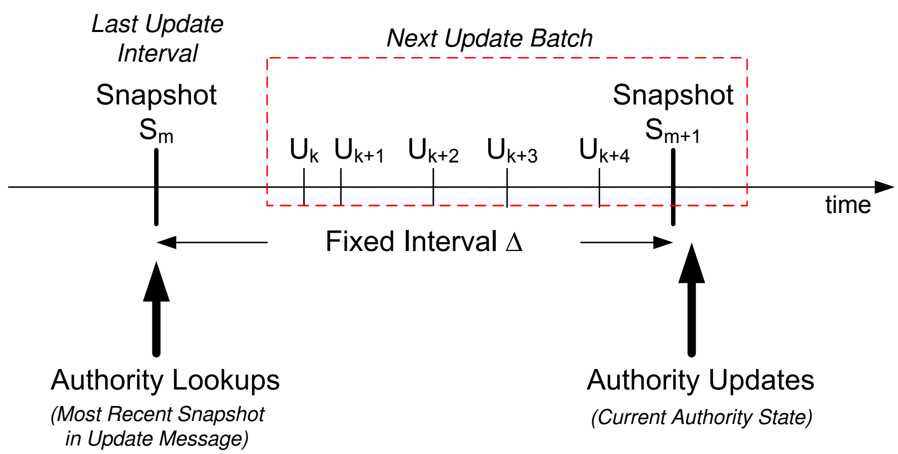

Figure 5-11: Updates and Snapshots Timeline

Updates

The authority state clients cannot be allowed to lag significantly behind, because this could permit the use of authority that has been revoked.

Our solution is to have the authority server send out periodic *updates*: each time it produces a snapshot, it also sends out an update message that contains the name of the new snapshot and describes all the changes from the previous snapshot to the one it just produced. Authority state clients then use this information to bring their caches up to date, so that they reflect the most recent snapshot.

Authority state clients do not process authority lookups if their state is "too old". They expect to receive an update message from the authority server every Δ seconds. If an update message doesn't arrive quickly enough, the authority state client stops processing authority lookups until it can communicate with the authority server.

Each update message contains a batch of entries from the operation log. As mentioned, for each operation, the authority server includes the type of the operation (e.g., `CreateTag`), the input and output values that are important for processing the update (e.g., the new TID generated), and plus the cores that are affected by this operation. The last entry of the update batch has the name of the database snapshot that the authority state client can fetch from after processing these updates locally.

When an authority state client receives an update message, it processes the message to reflect these changes locally.

The first thing the authority state client does is to discard all updates that do not affect any of the cores in its caches. We expect that what remains is a very small list, usually containing no updates. We expect updates to be rare (so that most update messages contain very few entries). Clearly the larger the deployment, the more updates in an update message, but in this case, we expect that many of the updates are for cores not cached locally at this authority state client.

For updates that affect cores in the core cache, the processing is simple. The core cache has in-memory tables that are similar in structure to the database relations at the authority server. For each update, the authority state client checks against the hashtable to see whether the affected core is cached; if so, it uses the information in the message to apply updates to the in-memory tables. The procedure is similar to what was done when the authority update was committed at the authority server. For example, if the update message specified that a `CreateTag` operation was committed, the entry from the update message will include the new TID, the core that it belongs to, and the principal that created it; the authority state client will add an entry to its in-memory `Tag Table` and also add an explicit delegation of this tag to the creator principal in the `Grant Table`.

For the content cache, processing of updates is more difficult. The problem is how to know whether a ⟨P, T⟩ pair in this cache is affected by an update. For example, in Figure 5-12, P_1 has authority for tag T because P_1 acts-for P_2, P_2 for P_3, P_3 for P_4, P_4 for P_5, and it is P_5 that has been directly granted authority for T. If the acts-for from P_4 to P_5 is revoked, and $\langle P_1, T\rangle$ is in the content cache, it must be removed since P_1 no longer has authority for T. Note that it is crucial to remove a pair if the principal is no longer authoritative for the tag. However, it would be safe to remove pairs that are still valid, since this cannot cause privilege to be granted erroneously.

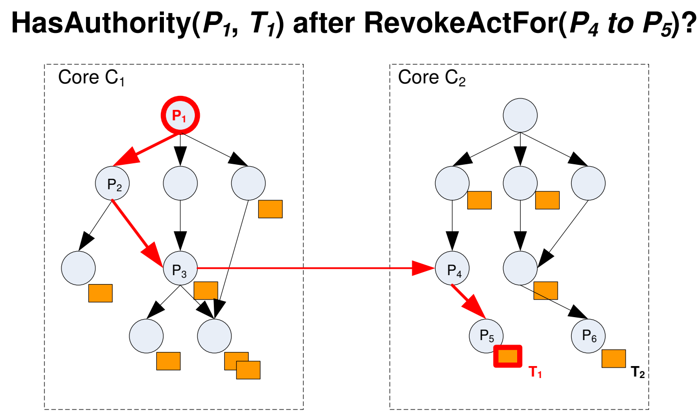

Figure 5-12: Problem with Invalidating the Content Cache

One possible solution is to simply clear the content cache. This is clearly correct but inefficient since the computation needed to place items in the content cache can be expensive. Our approach is to keep some information about the path used to place an item in the content cache. An entry ⟨P, T⟩ is placed in the content cache because a path exists from P to T and this path is determined by traversing the principal hierarchy in the core cache. Our system records the list of cores of the path that was found to provide this authority. For the example in Figure 5-12, the traversal of `HasAuthority(`P_1`, `T_1`)` uses cores C_1 and C_2 in the list of dependent cores. This path information is stored together with the $\langle P, T\rangle$ entry. With this information, the authority state client has sufficient information to know which entries to invalidate when it looks at the update message: if a content cache entry depends on a core C that has been modified, it is removed.

This approach will sometimes remove cache entries unnecessarily, for two reasons. First, even though one of the cores an entry depends on has changed, this change might not affect the entry in the content cache; for example, our approach will remove $\langle P_1$, T⟩ due to an unrelated update such as a revocation of T_2 from P_6. Second, the principal may have authority for the tag via a different path. For example, P_1 may act-for a principal in core C_3 that has authority for T.

To avoid discarding valid entries from the content cache, we could keep more detailed path information, at the level of principals rather than cores. We chose to use cores because we expect this to reduce the amount of needed metadata; each core along a path stands for a sub-path involving a number of principals local to that core. Also, even with detailed path information, we might still discard entries unnecessarily due to the second problem.

Inter-Node Consistency

In the preceding sections we discussed how compute nodes coordinate with the AS to assure that there are no inconsistencies. However, there are also issues involving communication between compute nodes. For example, suppose that node x already has received the version of the authority state that reflects Bob's move to company A. If code running at node x makes a call to node y, and y is using an earlier version in which Bob still works for company A, there can be a breach of the conflict of interest constraint.

We avoid this problem by including the current authority state version number of the sender in every message. When a node receives a message, it checks this number and if it is later than the authority state version being used locally, it communicates with the AS to bring itself up to date and it delays all requests to check authority until it is up to date.

Checking Validity of Principals

A problem raised by revocation of an act-for link is how to remove authority from processes running on the basis of the now-revoked link. For example, suppose Alice lets Bob act-for her, and Bob uses this authority in a long-lived process that is now running as Alice. Then Alice revokes the act-for link; the question is how to stop the long-lived process from proceeding based on authority that no longer exists.

Our solution is to maintain the "basis" of each USER-d's authority. Thus when Bob exercises the act-for link to start acting as Alice, this information is noted. Additionally, whenever a USER-d requests authority, e.g., to declassify, the USER-d passes the basis to the authority state client, and the authority state client checks the validity of every link in the basis; if any of these links has been revoked, it informs the USER-d that the process's authority is invalid and the USER-d is terminated. Additionally, the basis is sent in every message, so that we can recognize the revocation problem even when the revoked link was followed at a different node than where the process with that authority is now running.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Performance Evaluation and Optimizations

The Aeolus distributed computing platform is designed as a reference monitor that confines the execution of user applications. By interposing I/O and network requests, it ensures that information flow properties are preserved. However, this comes at a performance cost, as it adds overhead in serving these requests. We evaluate the overhead from various components of our platform, examining local operations as well as remote ones.

As an application executes, Aeolus platform instances require access to sufficient authority state to determine whether certain privileged operations should be allowed. For example, when an application running as principal P issues `Declassify(T)`, does principal P have authority for tag T? Our authority management architecture is designed to answer this question in a timely manner while presenting a consistent view of the authority state. We evaluate its effectiveness.

At the end of this chapter, we also include macro-benchmarks that examine the end-to-end impact of our platform on application performance.

6.1 Experimental Setup

The experiments described in this chapter are performed on a set of identical machines running the 32-bit edition of Windows XP Service Pack 3 with .NET Framework 2.0. Each system had 2.5 Ghz Intel Q9300 quad core processors, 4GB of RAM, and a 250 GB Western Digital 1601ABYS SATA/300 disk drives. The systems were connected via a 100Mbps switch, and the network roundtrip time is less than 1 millisecond. For most experiments, average values are computed over 1000 runs, at steady state of the system.

6.2 Local Platform Overhead

Aeolus loads user code into appDomains to restrict their access permissions and isolate their address spaces. Each platform instance consists of user appDomains that are used to run a user application, provide the necessary user code isolation, and a proxy object that interposes various requests in order to enforce the system's information flow properties. In this section, we look at the overhead introduced by our design choice and selected implementation techniques. First, we examine the cost of our isolation mechanism, comparing the cost of using application domains versus OS processes. Second, we look at the proxy overhead for communicating between user application domains (USER-d) and the system application domain (SYS-d).

6.2.1 Isolation Mechanism

The application domain is the unit of isolation in the Aeolus platform. Each USER-d runs with a principal. Computations may run with different principals and user code can be loaded and executed in different application domains, ensuring complete separation of program data between code running with different authority. In this section, we carried out an experiment to evaluate the basic cost of using an application domain versus an OS process.

This experiment measures the costs of loading an application, starting its execution and successfully terminating it, for two cases, one using an OS process and another using an application domain. Table 6.1 compares the time it takes to execute a dummy (no-op) application. The values presented are averages over 1000 runs.

Table 6.1: Execution Overhead of OS Processes and AppDomains

Isolation Type	Average Execution Time (s)
OS Process	0.117 370
AppDomain	0.012 510

Application domains experience roughly an order of magnitude less overhead in loading and unloading the code compared to OS processes (0.012s versus 0.117s). This is expected as application domains are language runtime abstractions where many application domains can exist in a single OS process. This substantially lowers the context switching overhead. Hence, application domains experience a much lower setup, tear-down, and inter-communication overhead.

This experiment provides an estimation of the performance difference one can expect between the two isolation mechanisms. Next, we evaluate how the use of appDomains impact the performance of Aeolus.

6.2.2 Inter-AppDomain Communication

Inter-AppDomain communication occurs frequently during the operation of the Aeolus platform. For each privileged call (e.g., access to I/O), the user application code cannot perform the operation directly since it runs in a restricted appDomain and must issue an inter-appDomain call from the proxy object to the SYS-d. This requires marshaling and un-marshaling the call from the restricted user application domain to the unrestricted SYS-d. In addition, when the call arrives at the unrestricted application domain, the .NET security model keeps the permissions of the executing code at the caller's (restricted) level and therefore, the SYS-d must explicitly raise the security permissions to handle the call and restore it to the restricted level after the call finishes. Table 6.2 compares the average roundtrip latencies of such cross-appDomain communication with an OS inter-process communication call. An OS IPC is roughly 8 times more expensive.

Table 6.2: Overhead of Cross Boundary Communication

Operation	Average Latency (s)
Inter-AppDomain Communication	0.000 025
Inter-Process Communication	0.000 200

6.3 Forks and Calls

6.3.1 Forks

As the application executes, programmers can use the Aeolus API to fork off pieces of code to run in a different appDomain with the same or a different principal. This is done by issuing `Fork` requests. Aeolus sets up a different appDomain for running the specified code with process labels reflecting the contamination of the caller.

Table 6.3 shows the average request time for executing Fork requests. When the same principal is used (i.e. `Fork`), the average request time is 0.458 ms. This time includes allocating an un-used appDomain from the appDomain pool in SYS-d and initializing it, running a simple `Echo` and returning the appDomain to the pool. Notice that this execution is substantially lower than the cost of using appDomain in Table 6.1 due to the use of the pool. A common case will be to switch to running code as P_{public}. The measurements show that this increases the cost slightly requiring 0.461 ms since the process state is updated.

In the case that a different principal is used, Aeolus must check that the requesting process is running with a principal that is authoritative for the principal that it is trying to switch to. The SYS-d consults the authority state client for this check, which requires an OS IPC call and some additional processing. `Fork to pid` requires roughly an additional 0.53 ms over `Fork to` P_{public} as a result of these additional checks. The measurements are taken against a hot authority client cache and with null process labels.

Table 6.3: Fork Overhead

Operation	Average Request Time (s)
Fork	0.000 458
Fork to P_{public}	0.000 461
Fork to pid	0.000 997

6.3.2 Calls

Application developers can also use `Call` to run methods with different principals. These methods are invoked in the same appDomain as the caller's. Aeolus checks that the operation is allowed and sets the process' labels appropriately, at the beginning and at the end of the call.

Table 6.4 compares a normal method call with one that switches to a different principal. The results show the average request times over 1000 runs of invoking a simple `Echo` method. Aeolus does not require special processing for normal method calls and these methods finish executing in less than 1 μs. For method calls that require a switch of principal, Aeolus does not require any authority checks when it is switching to P_{public} and the call suffers negligible overhead in this case. However, when it switches to other principals, Aeolus contacts the authority state client to check that the switch is allowed. In this experiment, we measure over hot authority state caches and so this check requires only local computation at the authority state client. Unlike a `Fork`, Aeolus must also update the process' labels at the end of the call. The average request time for `Call to pid` is 0.591 ms, roughly equivalent to the additional time between `Fork to` P_{public} and `Fork to pid` for the authority check.

6.4 Authority Closures

Authority closures are used to run code with an anonymous principal. We believe that they will be used frequently by applications to tightly control how authority for

Table 6.4: Call Overhead

Operation	Average Request Time (s)
Normal Call	0.000 000 3
Call to P_{public}	0.000 000 8
Call to pid	0.000 591 6

certain tags are used.

When an authority closure is called, Aeolus must first determine if the call is legal; it needs to check that the hash of the specified code is indeed signed by the expected private key. In the previous chapter, we describe our use of GAC and strong name to speed up this check. Recall that a closure ID is bound to a private-public key pair and an anonymous principal. When the closure call is issued, Aeolus must use this ID to lookup the key and anonymous principal. We have a mapping of this binding in the SYS-d to reduce communication with the authority state client. Here, we use a simple micro-benchmark that generates a set of application calls that invoke authority closures on our platform. Table 6.5 presents the average execution time for running a simple `Echo` method inside an authority closure. With all these optimizations, a closure call takes 0.76 ms, longer than a `Fork to` P_{public} (as it still requires executing in a new USER-d) but less than `Fork to pid` (due to these optimizations).

Table 6.5: Cost of Executing Authority Closure

Operation	Average Execution Time (s)
Closure Call	0.000 760

6.5 Remote Procedure Calls

In this section, we evaluate the overhead of a remote procedure call issued within our platform. Aeolus offers remote procedure call via web service calls.

Our benchmark issues the same type of web service call on both a normal .NET web service implementation and one that has been wrapped by an Aeolus service call proxy and deployed on our platform. The application benchmark is a simple `Echo` web service call that causes an input string to be sent to the server and then sent back to the client. This measures the round-trip latency of a web service invocation. We took the average of 1000 runs and reported our measurements in Table 6.6.

Table 6.6: Interposition Overhead of Web Service Invocations

Web Service Type	Average Latency (s)
Base Web Service	0.002 138
Aeolus Web Service	0.003 600

The `Aeolus Web Service` version measures the end-to-end time from the remote call invocation in the user application to the time the result is returned to the application. This includes the time for the request to cross the USER-d-SYS-d locally, the packaging of the information flow state header in the SOAP message, and the dispatching and setting up of an appropriate appDomain for running the actual method in the user web service library on the server. And of course, this also includes the reverse path for the result to propagate back to the client. We have found that in optimizing our web service performance, the appDomain pool played an important role as this significantly cuts down the setup cost in executing the user web service logic on the server. All the necessary libraries and objects were instantiated and made ready-to-go in these pre-loaded appDomains. Only the information flow state needs to be initialized at the time of the call.

In our experiment, the web service client and server are located on different machines. The network latency is negligible. For each measurement, 1000 web service invocations are issued consecutively. From Table 6.6, `Base Web Service` took on average 2.1 ms to service a request compared to 3.6 ms with `Aeolus Web Service`. This is roughly a 1.7X slowdown. Further investigations show that about 1.2 ms of this time is spent at the server, which shows that the SOAP message interposition overhead is small.

This benchmark stresses the measured system as the network latency in real deployments are often far greater than several milliseconds making the Aeolus overhead much less significant in actual applications. Furthermore, web service latency can be hidden by the common design practices (e.g., by making these calls non-blocking so that the application can move on to other tasks before the results are ready). Given these factors, the observed overhead is unlikely to impact application performance.

6.6 File System

Aeolus FS manages metadata to keep track of the sensitivity of files and directories and checks incoming file requests against them. It does this by keeping a database relation that has an entry for each file and directory recording the filepath, filename, secrecy label and integrity label. In addition, there is a root entry that protects the entire file system.

Aeolus FS imposes a level of indirection between the user application and the actual file system implementation. Applications making use of an Aeolus FS will experience overhead in label checking and in the management of related metadata. In this section, we investigate this performance overhead, by comparing it to the native commercial file system running below our platform.

Applications running on Aeolus are likely to execute and store only temporary data on compute nodes, most of the persistent data will be stored in networked file systems on storage nodes. Hence, in this experiment, we compare the case where the client (i.e., benchmark application running on a compute node) and the storage node are on separate machines. We contrast the implementation of a file system storage service served by a native file system versus one that we have designed to store sensitive data. In the latter, there is an Aeolus FS wrapper that manages its own metadata in a database while storing file content in the native file system. The former simply provides a remote interface to the native file system. In our experiment, the native file system is Windows NTFS.

In the first experiment, we use a benchmark that generates a set of directories and files at various depths and with varying number of files in each directory. The maximum directory depth is set at 5 and there can be a maximum of 10 files in each directory. The number of tags in secrecy label and integrity label vary from 0 to 6. The benchmark is directed at the two implementations discussed, on the native file system, `Native FS`, and on the Aeolus FS file system, `Aeolus FS`. Table 6.7 shows the request execution time for different file system administrative operations. The values in this table represent the average taken over 1000 requests. When files and directories are created, not only does our system need to check whether the requester has acceptable secrecy and integrity labels to permit the operation, the Aeolus FS wrapper also has to update its internal tables to keep track of the labels of the new files and directories. These contribute to the observed overhead. With the exception of `RemoveDir`, most of the overhead is well below 2X. These administrative operations do not occur too frequently over the lifetime of an application and hence the observed overhead is quite acceptable. Our implementation of `RemoveDir` recursively removes all sub-directories and files requiring removal of all related metadata, the native file system may have optimizations to perform their recursive removal quickly or lazily.

Table 6.7: Request Servicing Time for File System Administrative Operations

FS Operation (in s)	Native FS	Aeolus FS	Overhead Factor
CreateDir	0.001 230	0.001 573	1.28X
CreateFile	0.001 822	0.002 466	1.35X
ListDir	0.001 211	0.001 418	1.17X
RemoveFile	0.001 636	0.002 721	1.66X
RemoveDir	0.001 300	0.004 194	3.23X

Next, we run experiments to measure the time it takes to read and write files of varying sizes. Through the Aeolus file system API, there are two ways in which user applications can read and write files. In the first method, they can do this by reading and writing the entire file, using `ReadFile` and `WriteFile`. In the second method, they can use file-streams to read and write files in chunks at various points during an application execution. The file-stream method is preferable especially for large files where the application can work on one part of the file and later on another part. Also, it is desirable for appending to a file, for example, when the application is using a file as a type of status log. We measure the time for reading and writing a file at a directory depth of 5 for 1000 runs.

Table 6.8 and Table 6.9 show the results for `ReadFile` and `WriteFile`, respectively. Both the Native FS and Aeolus FS use the same function to store the file content, namely, .NET's `IO.File.ReadAllBytes()` and `IO.File.WriteAllBytes()`. These methods are synchronous and automatically flush the file content at the end of the function execution. For very small files, at 10KB file size, Aeolus FS experiences the greatest overhead as the time taken for label checking operations is relatively significant compared to the time taken for the file content I/O operations. Even in this case, the Aeolus FS has only a less than 3% overhead for file reads and writes. At 10MB, this overhead is almost negligible at 0.02% and 0.09% for reads and writes, respectively.

Table 6.8: ReadFile Request Time for Files of Different Sizes

Average Request Time (s)	Native FS	Aeolus FS	Overhead Factor
10KB	0.005 5	0.005 6	1.022 2
100KB	0.021 9	0.022 2	1.010 6
1MB	0.188 7	0.188 2	0.997 0
10MB	1.866 1	1.866 5	1.000 2

Table 6.9: WriteFile Request Time for Files of Different Sizes

Average Request Time (s)	Native FS	Aeolus FS	Overhead Factor
10KB	0.004 6	0.004 7	1.026 2
100KB	0.020 8	0.021 0	1.012 2
1MB	0.194 4	0.193 7	0.996 8
10MB	1.989 8	1.991 7	1.000 9

We also measure file-streams. The last two tests measure the cost of sequentially reading or writing a file using a file-stream. Most of the overhead is in the label check that occurs when the file is first used. When data is read or written, we only need to do the checks if the file-stream is marked as needing to be checked. The experiments assume that the check isn't needed, since this is the normal case. The check is needed only if the process has changed its principal or labels since the last use of the file-stream making it questionable as to whether the file-stream is still valid. The file is read or written in 4KB chunks since this is the default size of the underlying system. Table 6.10 shows that Aeolus introduces very little additional cost over the native file system for reads and writes. Writes are relatively cheaper in both cases because they are flushed asynchronously.

Table 6.10: File-stream Open, Read and Write

Average Request Time (s)	Native FS	Aeolus FS	Overhead Factor
Open	0.001 4	0.001 6	1.20
Read			
10KB	0.004 7	0.004 9	1.04
100KB	0.040 3	0.040 7	1.01
1024KB	0.415 9	0.416 0	1.00
Write			
10KB	0.003 2	0.003 2	1.00
100KB	0.026 7	0.026 7	1.00
1024KB	0.272 4	0.272 7	1.00

6.7 Boxes

Boxes provide encapsulation of sensitive data, preventing unnecessary contamination prior to the viewing of the box content. Table 6.11 shows the time it takes for basic operations such as `Put` and `Get` on an `AeolusBox`. In this experiment, the process and box labels are empty and the box content consists of a simple integer value measuring the raw overhead of such calls. `AeolusBox` is implemented as an abstract data type and hence, no cross-appDomain communication is required and these basic operations are very fast, taking only 18 us.

Table 6.11: Cost of Basic Operations on Boxes

Box Operation	Average Request Time (s)
Put	0.000 018
Get	0.000 018

6.8 Shared State

While Aeolus FS shows very low overhead over the native file system, sharing of local data over network filesystem is not very efficient. Aeolus provides shared objects for local data sharing and shared queues and shared locks for synchronization between applications running on the same machine.

6.8.1 Shared Objects

In this experiment, we measure the cost of basic operations on a shared object that contains an integer. The shared state manager is implemented in the SYS-d and accesses to shared objects require inter-appDomain communication. Table 6.12 shows that accessing this shared object takes 0.05 ms. This includes the label checks, serialization and de-serialization of the object across the appDomain boundary. `ReplaceObject` is slightly faster than `GetObject` since it is a unidirectional request.

Table 6.12: Cost of Basic Operations on Shared Object

Shared Object Operation	Average Request Time (s)
GetObject	0.000 054
ReplaceObject	0.000 051

We also vary the object size and Figure 6-1 shows the average request times for different object sizes. We observe a negligible increase in service time for this range of object sizes.

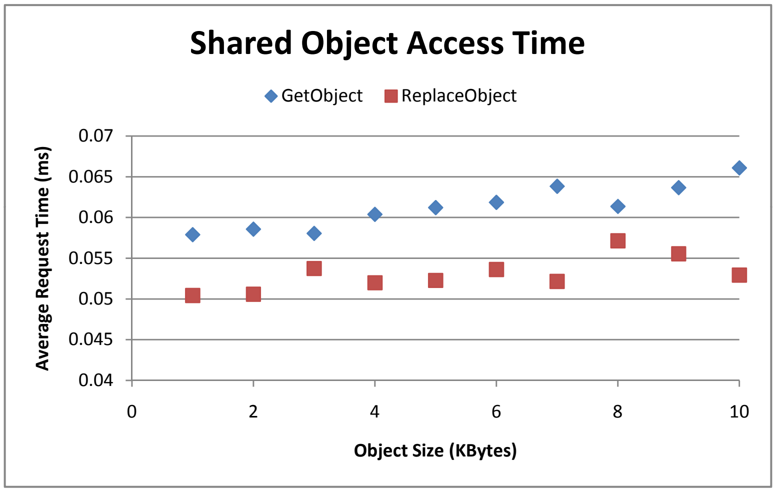

Figure 6-1: Shared Object Access Time as Object Size Varied

6.8.2 Shared Queues

A process can also communicate with another process via shared queues. We describe two experiments in this section. First, we show the costs of basic operations on a shared queue. Then, we show the results of an experiment that mimics an IPC call between a client process and a daemon process using shared queue as the underlying mechanism.

We present the cost of enqueue and dequeue operations on a queue that stores integer-size messages in Table 6.13. The experiment is setup such that these operations are served immediately (e.g., `WaitAndDequeue` is issued on a non-empty queue). These operations take less than 0.1 ms. `WaitAndDequeue` takes slightly longer than `Enqueue` due to the blocking nature of this call.

Table 6.13: Cost of Basic Operations on Shared Queue

Client Request	Average Request Time (s)
Enqueue	0.000 053
WaitAndDequeue	0.000 061

In the second experiment, we have two processes. The daemon process has a request queue and the client process has a reply queue. The daemon process is a listener waiting for incoming messages in its request queue. The client starts by enqueuing a message into the daemon's request queue, then it issues `WaitAndDequeue` to wait on a message from its reply queue. Upon receiving a message in the request queue, the daemon sends back a reply by enqueuing a response message on the client's reply queue. Table 6.14 measures the costs of these operations from the client process. The average `Enqueue` time is 0.053 ms. The `WaitAndDequeue` request time includes issuing the request on the shared state manager, alerting the daemon process with the incoming message, processing by the daemon to generate and enqueue the response message on the client's reply queue, and finally alerting the client process. Hence, we see that the `WaitAndDequeue` time is longer requiring on average 0.319 ms to get a reply from the daemon with this setup. The IPC roundtrip time is therefore 0.372 ms.

Table 6.14: IPC using Shared Queue

	Average Request Time (s)
Enqueue	0.000 053
WaitAndDequeue	0.000 319
IPC roundtrip	0.000 372

6.8.3 Shared Locks

While shared queue can be used to implement locks, Aeolus provides a more direct locking mechanism via shared locks. Table 6.15 shows the average request time for the `Lock` and the `Unlock` operation on a shared lock when there is no contention. The cost of these operations consists primarily of the inter-appDomain calls to shared state manager in SYS-d and the label checks.

Table 6.15: Basic Operations on Shared Lock

Shared Lock Operation	Average Request Time (s)
Lock	0.000 054
Unlock	0.000 051

6.9 Authority Management

In this section, we examine the performance of our authority management scheme by studying its sub-components through a set of micro-benchmarks.

We quantify the costs of various micro-operations on our authority state client caches. First, we compare the reduction in request latency resulting from the content cache and core cache. Then, we measure the cost of eviction in both caches. Lastly, we study the cost of processing authority update messages and the cost of invalidations.

6.9.1 Cache Component Latency

We evaluate our design choice of using a two-tier cache and quantify the benefits of caching authority state. We compare the average authority lookup latencies of three configurations: `Content Cache only`, `Core Cache only` and `Remote AS`. `Remote AS` refers to the case where all requests go directly to the remote authority server and the two-tier cache is not used at all. The AS determines the answer by running a number of queries against the central authority state database and returns an answer of either `yes` or `no`. For the other two cases, authority checks are done locally at the authority state client. The experiment preloads the caches and hence there is no communication to the remote AS at all. In the `Content Cache only` case, all requests are served by a hot content cache. In the `Core Cache only` case, the fast content cache is disabled and all requests are served by a hot core cache.

This experiment consists of generating authority requests that traverse a varying number of principals and cores. It creates simple linear delegation chains in the principal hierarchy without any branching. Exactly one principal is created in each core and delegation chains are created by giving act-for privileges to principals in other cores. For example, a chain of length 5 involves 5 principals over 5 cores connected by 4 act-for links and 1 tag delegation link. The principal at the head of the chain acts-for all principals later on in the chain. The last principal in the chain has explicit authority for a tag and our experiment measures the request latency in asking about whether the principal at a certain position on the chain has authority for the tag at the end of the chain.

We run 1000 authority requests for each of these configurations. Both caches are sized large enough such that there are no evictions. There are also no authority update messages causing updates or invalidations in the measured time interval. Figure 6-2 and Table 6.16 show the results of this experiment.

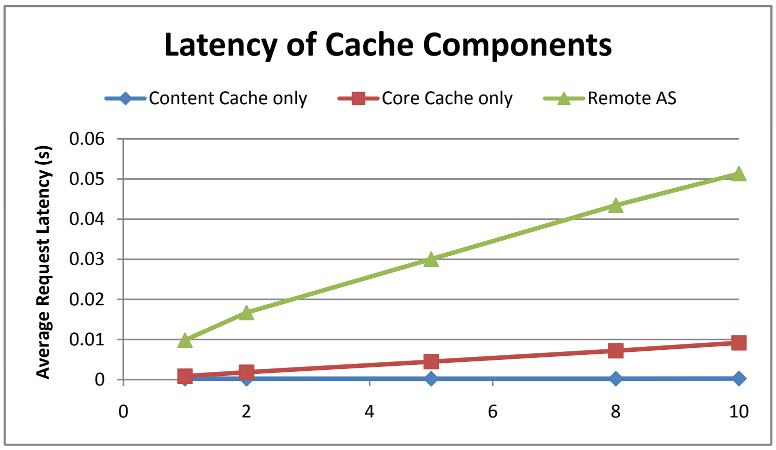

Figure 6-2: Latency of Cache Components

Table 6.16: Speedup using Authority Caches over Remote AS

	Delegation Chain Length				
	1	2	5	8	10
Speedup					
Content Cache only	46.76	79.52	143.00	206.90	205.36
Core Cache only	11.55	9.13	6.72	6.05	5.60

The results show that even having just a core cache provides a large speedup over consulting the AS; note that costs presented here for using the AS are probably lower than they would be in a real deployment since our AS is close to the node making the request, and is unloaded. Furthermore the table shows that the content cache provides another speedup over the core cache. The content cache takes roughly the same length of time to answer the query regardless of the length of the chain. This is expected since the authority check is answered by a simple hashtable lookup. For the AS and the core cache, the time is linear in the length of the chain since their request handling time is dependent on the amount of data they are computing (or traversing) over.

The most interesting column in the table is actually the one for chains of length 1. This represents the best case for both direct use of the AS and for the core cache; still the content cache does substantially better. Furthermore we expect this case to be a common one because a good methodology for applications is to run a process with the principal that has direct authority for the tag, since this is in accordance with the principle of least privilege.

6.9.2 Eviction

To assess the costs of evictions, we first measure the additional latency that results from eviction in the content cache by comparing two phases of an execution run. We generate 20,000 independent authority requests, that is, different principals in different cores asking about authority for different tags (with a delegation chain length of 1). We set the core cache size to a very large value while limiting the content cache size to 10,000 entries. We start the experiment with both caches enabled but empty. In the first phase, 10,000 of these requests are issued with all of them missing in both caches and causing new entries to be added. There are no evictions in the first phase. In the second phase, another 10,000 requests are issued. This time, the content cache is always at its maximum capacity and every request causes not only both caches to be filled but also an entry to be selected and evicted from the content cache. As shown in Table 6.17, the difference in request latency between the two phases gives the cost of evicting an entry from the content cache, which is about 0.3 ms. This eviction cost includes updates to LRU data structures and metadata that keep track of the cores that each content cache entry is dependent on.

Table 6.17: Content Cache Eviction Cost

Phase	Average Request Latency (s)
1. Fill-only	0.010 947
2. Evict-and-Fill	0.011 240

We also run a similar experiment to measure the core cache eviction cost. We divide the execution into two phases and set the content cache size to a large enough value such that there are no evictions in content cache. In the second phase, core cache evictions occur for every request. This causes a core cache entry to be selected and evicted. Table 6.18 shows the cost of evicting a core from the core cache is roughly 0.4 ms based on the difference in request latency between the two phases. This cost includes selecting the core to be evicted using the LRU data structures and going through all the in-memory tables of the core cache and removing the entries that belong to this core. The eviction costs for both content and core cache are well below 1 ms. These costs are primarily due to the data structures used and we intend to optimize this in the future.

Table 6.18: Core Cache Eviction Cost

Phase	Average Request Latency (s)
1. Fill-only	0.010 846
2. Evict-and-Fill	0.011 248

6.9.3 Processing of Update Messages

In this experiment, we want to understand the combined cost of processing authority update messages, updating the core cache and invalidating the content cache.

Authority update messages are processed as a batch once in each update interval. We study this invalidation/update processing time as a function of the number of cores that are invalidated in each batch of messages. This batch processing time includes the time to parse the messages, perform updates on the core cache, and invalidate appropriate content cache entries.

We use the above micro-benchmark with a delegation chain length of 1. That is, each chain has exactly one principal and each principal has been explicitly delegated one tag. Each principal is in a different core. We perform 1000 runs for each experiment with different number of cores in each message batch. In each run, we

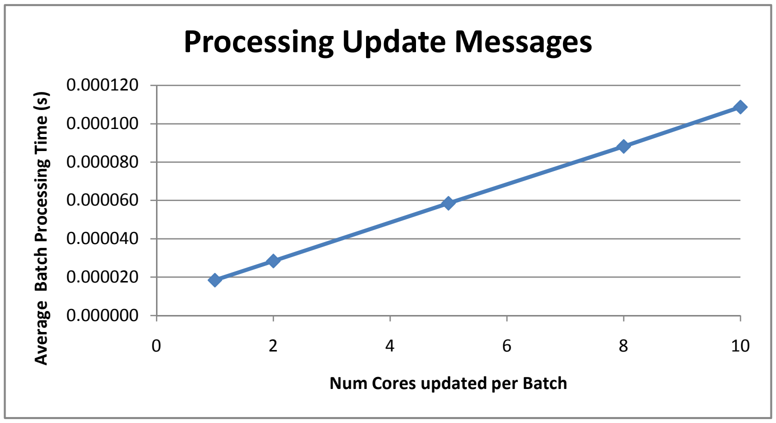

Figure 6-3: Cost of Processing Authority Updates

first generate a set of authority lookups that affect the desired number of distinct cores. Then, we issue updates to the AS to remove the delegations that these lookups depend on, causing the same number of cores to be affected. Stream broadcast is triggered and the authority state client is informed of these changes. When the authority state client receives these messages, it performs the necessary content cache invalidations and core cache updates. Figure 6-3 shows this processing time as we vary the number of cores affected in each experiment. As expected, the processing time increases almost linearly with the number of cores affected since the number of entries updated/invalidated increases proportionally.

6.9.4 Miss Penalty

The miss penalty an application experiences depends on a number of factors such as the length of the delegation chain, the amount of authority state that needs to be fetched in order to answer the lookup request and how the authority state is organized by the application or organization. We do not present a comprehensive study in this section but merely conduct a simple experiment that gives a sense of how request latency changes with variation in miss rate.

In this experiment, we use the benchmark in Section 6.9.1 and set the delegation chain length to 10. We use a content cache size and a core cache size of 10,000. The average latency is measured over 10,000 authority requests. First, we fill these caches

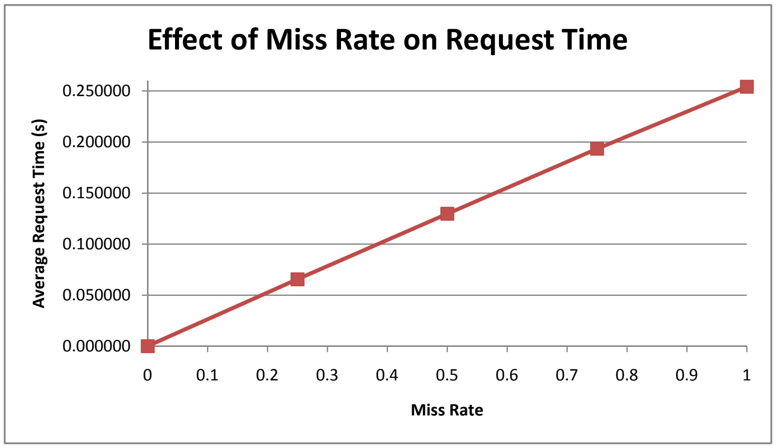

Figure 6-4: Effect of Miss Rate on Request Time

with entries according to the miss rate we want to measure. For example, to get a miss rate of 25%, we issue 7500 authority lookup requests (each of these requests causes one entry in the content cache and one entry in the core cache to be added). Then, we issue 10,000 additional requests which includes repeats from the ones issued previously (in the case of 25% miss rate, that translates to 7500 previous requests and 2500 new requests). Figure 6-4 shows the results of the measurements. Since all authority requests depend on the same number of principals, tags and cores, the cost of a miss is constant and as the miss rate increase, the average request latency increase linearly.[1] The cost of a miss includes the time to retrieve core from the remote AS, populate the core cache, compute the result of the authority lookup, and populate the content cache.

6.9.5 Cache Sizes

The content and core cache sizes change depending on the entries that are stored even though the maximum number of entries are fixed. For the content cache, the lookup cache has the following fields in each entry: principal ID, tag ID, and a list of core IDs that this entry depends on. The pair ⟨principal ID, tag ID⟩ are used as keys for the cache lookup. The list of core IDs are used for managing evictions

[1]It is worth pointing out that at a 100% miss rate, the average latency in this figure is higher than the `Remote AS` case in Section 6.9.1 since entire cores need to be transferred over the network before the authority state client computes the result.

and invalidations. Table 6.19 shows the maximum content and core cache sizes for the micro-benchmarks generated in this section. The maximum number of entries in content cache and core cache are both set to 10,000. The maximum cache sizes are shown for different chain lengths. In these sets of benchmarks, more principals and more delegations are involved as the number of principals in a chain increases. Looking at the content cache results, this leads to more cores that each cached entry is dependent on and therefore, a larger cache size as the chain length increases. For the core cache, each entry represents all the relevant authority information for one core, that is, the subprincipals, tags created by these principals, acts-for delegations and explicit grants given to these principals. For chain lengths of 10, the content cache is less than 1MB while the core cache is less than 7MB. There are 100,000 principals and 100,000 cores in this case. This gives an estimate of the size of authority information one may expect.

Table 6.19: Cache Sizes

	Number of Principals in Chain				
Cache Size (KB)	1	2	5	8	10
Max Content Cache Size	234.38	312.50	546.88	781.25	937.50
Max Core Cache Size	1015.75	1640.75	3515.75	5390.75	6640.75

6.10 End-to-End Evaluations

In this section, we evaluate the overall performance of Aeolus applications. We chose two applications for our study: online store and secure wiki.

6.10.1 Online Store

This application mimics an online store and is illustrative of a very common application pattern involving web services. Many web services are frequently invoked to perform short client computations and exhibit session-like access pattern. In this experiment, the online store web service serves many users. Users make calls on the server, for example, to update their shopping cart or to finalize their purchase. Each web service call is invoked on behalf of one user. Each user is represented by a unique principal. We assume that the authority state of these principals is very compact, so when principals perform multiple operations as part of a single session, the relevant parts of the authority state will already be in the content cache.

In this experiment, we explore the performance of web service calls looking at how the request time varies with the workload. The workload parameter reflects the temporal locality of principals' use of privilege. We vary the number of new principals encountered in a measurement run to study different session access patterns. We measure the latency of a simple web service call that receives a client request, performs a computation (i.e., compute the total amount in a shopping cart), and then returns a 4KB result (i.e., HTML page) to the client.

The results are presented in Table 6.20. For a normal web application implementing this web service, the average request latency is 16.67 ms. The Aeolus-enabled version calls a closure; the closure adds a tag to its secrecy label; does the computation; declassifies by removing the tag; and then returns the 4KB result. Thus, the latency includes a remote method invocation, a closure call, a safe label manipulation, and a privileged label manipulation. Each call is made with a different principal. When all these principals belong to a session, the average request latency is 19.90 ms. The data show that even if locality is relatively poor, forcing Aeolus to fetch cores from the AS, the additional overhead imposed by Aeolus is small, requiring 22.31 ms when all requests run on behalf of new principals.

Table 6.20: Average Request Service Time of Online Store Web Service

% New Principals	0	25	50	75	100
Normal Web Application (in s)	0.016 67				
Aeolus Web Application (in s)	0.019 90	0.020 51	0.021 13	0.021 77	0.022 31

6.10.2 Secure Wiki

We also took an open-source web application, ScrewTurnWiki [65], and extended it with security features using Aeolus. We protect against the unintended disclosure of user passwords by storing them in a labeled file and allowing only an authority closure invoked during user log-in to access and declassify this information. We also confined the execution of a third party plugin, the Basic Statistics plugin, to prevent it from arbitrarily disclosing page access logs. The next chapter discusses the details of these modifications. In this section, we evaluate the performance overhead as a result of these modifications. We do this by comparing the latency of the unmodified ScrewTurnWiki implementation with our modified version.

In the first experiment, we measure the time it takes to process a user log-in request at the server. The Aeolus version retrieves the labeled file containing the user password and compares it to the one entered. The comparison is done in an authority closure. Table 6.21 shows that the original ScrewTurnWiki took 8 ms to serve the request whereas Aeolus required 12 ms. The additional overhead is primarily the cost of running the checking code in an authority closure and the overhead of using Aeolus FS on the server machine rather than using the local filesystem.

Table 6.21: ScrewTurnWiki Login Server Processing Time

	Base	Aeolus
Average Processing Time (s)	0.008	0.012

In the second experiment, we enable the Basic Statistics plugin and measure the client request latencies. This plugin is invoked prior to returning a wiki page to the client. In the Aeolus version, this plugin is executed on top of our platform, runs with the P_{public} principal, and appends data to a labeled file. Table 6.22 presents the average request times for the original ScrewTurnWiki and for the Aeolus version. Using Aeolus resulted in a slowdown of 1.35X in this case.

Table 6.22: ScrewTurnWiki Page Fetch Latency with Statistics Plugin enabled

	Base	Aeolus
Average Request Time (s)	0.0101	0.0136

Chapter 7

Application Case Studies

In this chapter, we study the design and development of information-flow-aware applications using the Aeolus programming model. We designed two very different applications to put ourselves in the application developers' perspective, exploring how one would assign tags to data and organize authority with principals. We also tried to see where it will be natural to use our programming mechanisms such as authority closures, boxes and shared state.

The first example is the Retail Kiosk. This illustrates a distributed computation that involves software providers from different organizations that need to protect their proprietary data while still being able to provide software services and collaborate with other vendors. The client logs on to a kiosk at a store and this triggers actions at five different web servers. The second example is the Clinical Medical Research System, which shows a very different scenario of how information flow can be used within a large organization that has a broad range of personnel. Electronic medical records are used by physicians as well as staff responsible for generating medical bills and the disclosure of data derived from these records is carefully controlled.

In addition, we also explored the use of Aeolus to improve the security of an existing application. This application is a wiki server that is accessed by many users and supports third-party plug-ins. We extended an existing application, ScrewTurn-Wiki, using Aeolus to provide new security features. We found that we were able to improve the security of this server by adding only a few lines of code and using a simple methodology in which sensitive information was moved to our file system and controlled through closures.

7.1 Retail Kiosk

This Retail Kiosk case is chosen as an illustration of some of the complex enterprise processing and highly integrated backend computation that drive today's businesses. The Retail Kiosk is targeted at the store-of-the-future where kiosks are used to deliver better customer services. The scenario we use is extracted from studying a real-world platform being developed at HP [37, 38].

7.1.1 Application Scenario Description

The Retail Kiosk provides various appealing functionalities. In this section, we follow a particular one: the retrieval of personalized coupons. As a customer approaches this kiosk at a store, the system displays a set of in-store coupons tailored to this customer. A single task like generating coupons can involve many software services to determine what products should be on promotion, to manage customer relationships and privacy concerns, and to run statistical models to predict what products may appeal to a specific customer. A lot of sensitive information can be collected and used from different storage systems and by enterprise services belonging to different organizations. Third-party software vendors may provide businesses with management suites. For example, the `Promotion Service` and the `Sales Analyzer Service` are aimed at helping retailers decide what products make strategic sense to be promoted based on the current inventory status of a retailer and industry-wide trends. These services can be invoked by different retailers. In order to generate coupons tailored to a particular customer, the services need to retrieve some information about the customer. The `Customer Management Service` maintains and manages collected customer information according to some privacy policies. A `Behavior Modeling Service` uses the customer information and its data models to classify the customer's shopping behavior. All this information is then used to determine the final set of coupons to offer. Figure 7-1 shows the interactions between these services. The arrows indicate the direction of information flow.

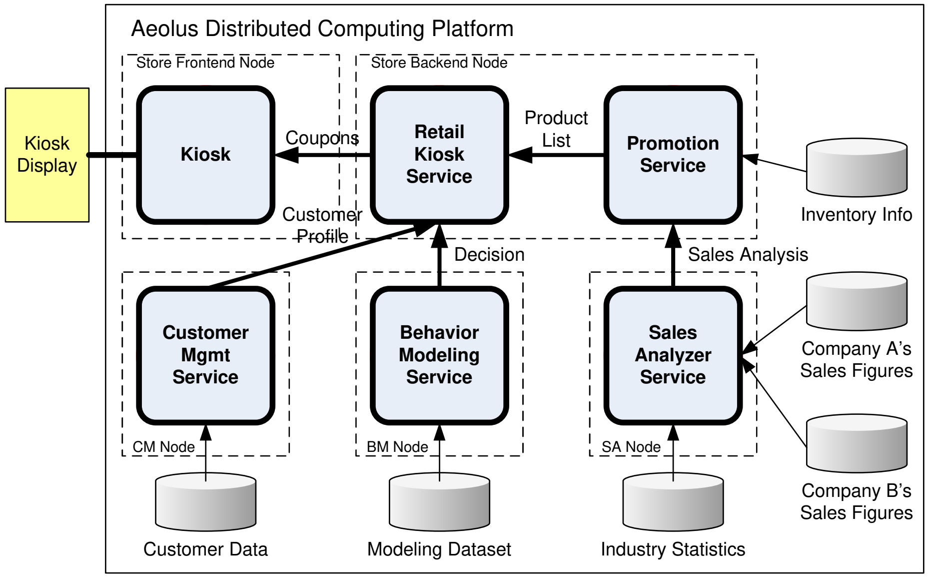

Figure 7-1: Retail Kiosk Scenario

This application scenario demonstrates an application solution that results from the interactions of five organizations providing software services on different servers. The `Kiosk` runs as a local application on a display kiosk while the `Retail Kiosk Service` runs on an in-store backend server. Consulting firms offer the `Promotion Service` and the `Sales Analyzer` as subscriber-based software exposed as web services. The `Behavior Modeling Service` and `Customer Management Service` are provided by different software vendors.

7.1.2 Data Security Concerns and IFC Program Structure

The `Kiosk` application manages a user session at the retail store. A user logs on and authenticates to the system. The `Kiosk` application ensures any data displayed to the customer must belong to the customer or be properly declassified. For example, data with a retailer's sales performance sensitivity cannot be disclosed. This is achieved by running the `Kiosk` application with the principal ID associated with the customer (e.g., P_{Alice}). In this way, it has the authority to declassify and endorse only tags that are created by this principal. The `Kiosk` sits at the boundary of our system and works together with an authentication mechanism to provide a secure channel for displayed data (outside of our system).

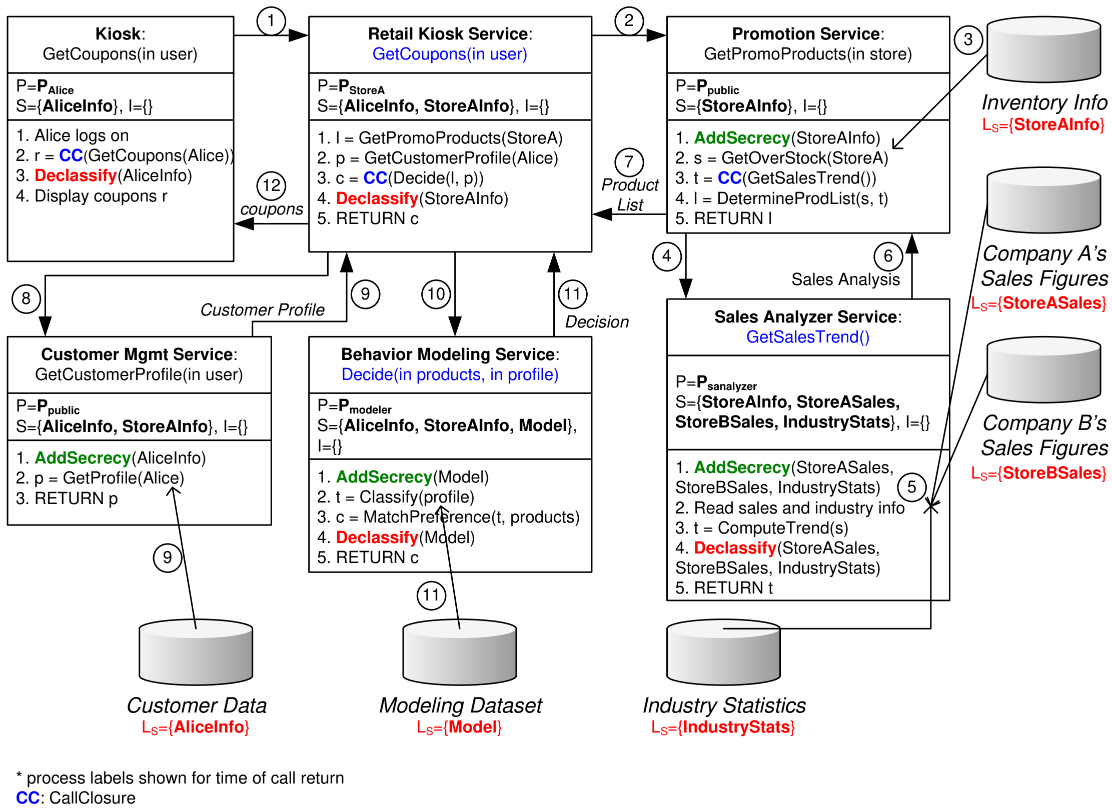

Figure 7-2: Retail Kiosk implemented using Aeolus

The `Kiosk` application is driven by backend logic such as the `Retail Kiosk Service`. It calls the `Retail Kiosk Service` to do the actual retrieval of coupons. The `Retail Kiosk Service` also belongs to the store and runs as part of the retailer's backend system. A customer can invoke these backend services but the store definitely does not want a customer to have the authority for the tags used to protect the store's inventory data. When a `GetCoupons()` call is made on the backend service, it is called as an authority closure to decouple the authority. A new computation is created to run this code under the anonymous principal P_{StoreA}.

The `Retail Kiosk Service` makes several calls to remote services that assist in coming up with the final list of coupons. First, the `Retail Kiosk Service` invokes a `Promotion Service` to analyze the store's inventory status and to figure out what products should be advertised. The `Promotion Service` does not need any authority and hence, it is run with the principal P_{public}. Prior to accessing the inventory for `StoreA`, it needs to add the `StoreAInfo` tag to its secrecy label to permit the read. This service is further decomposed into other services. A `Sales Analyzer Service` is designed to come up with popular product categories based on different companies' sales figures. Therefore, how it uses and discloses this information must be carefully controlled. Sales data and industry trends are marked with tags in their secrecy labels. For example, Store A uses the tag `StoreASales` to protect its sales figures. The `Sales Analyzer Service` must be executed as an authority closure. Only the anonymous principal $P_{sanalyzer}$ has sufficient authority to declassify individual companies' sales data and this principal can only be used with the code `GetSalesTrend()`. Different stores trust this authority closure to declassify their sales figures. When a company enrolls with the `Sales Analyzer Service`, it grants this closure authority for its tag. Based on the sales trend and the inventory status, the `Promotion Service` returns the promotional product list to the `Retail Kiosk Service`. The `Retail Kiosk Service` becomes tagged with an additional `StoreAInfo` tag as a result of this.

Next, the `Retail Kiosk Service` invokes a `Customer Management Service` to retrieve the customer's profile and in the process, gets marked with the customer's tag (i.e., `AliceInfo` in its secrecy label). Using the customer's profile and the product list, the `Retail Kiosk Service` consults the `Behavior Modeling Service` about which products on the list should be offered based on this customer's profile. The `Behavior Modeling Service` is administered by a different organization and uses a proprietary modeling dataset to make these decisions. This dataset is protected by the tag `Model` created by this organization. The `Behavior Modeling Service` is presented as an authority closure so that this organization can have full control on how any dependent result is declassified. The authority for the `Model` tag is bound to an anonymous principal $P_{modeler}$ which is associated with the `Decide()` code. This code classifies the customer based on his/her profile and selects products from an input list that will be appealing to the customer using its proprietary algorithms.

Now, the `Retail Kiosk Service` has a list of coupons tailored for the logged on user. Its process has the two tags `StoreA` (from the `Promotion Service`) and `AliceInfo` (from the `Customer Management Service`) in its secrecy label. Prior to returning this list to the `Kiosk` application, it removes the `StoreA` tag. The `Kiosk` application can then remove the customer's tag and display the coupons to the user standing at the kiosk.

7.1.3 Programming Experience

Using the Aeolus programming model, it was easy to think about the authority needed for different code segments. The explicit safe label manipulations helped identify code that consumes sensitive data and from there on, we can consider the different outputs that are derived from it and under what circumstances might declassification be acceptable. For example, the product list generated by the `Promotion Service` is tagged with the `StoreA` tag after it has learned about the inventory status of the store. We do not want to declassify this data immediately and by keeping it tagged, we prevented the `Behavior Modeling Service` from leaking it. Instead, when the `Retail Kiosk Service` has narrowed this down to a much shorter list of coupons that will be generated, we declassify the `StoreA` tag.

We also found that authority closures were very useful especially in this case where many different organizations are involved. They do not trust each other enough to use delegations. Notice that the principal hierarchy is very bushy, with principals (and anonymous principals) being directly authoritative for tags. Authority closures are preferred in this scenario since they allow these organizations to better limit how authority can be used. We saw that this was useful in two very different cases. The `Behavior Modeling Service` uses an authority closure to run its own code and protects most of the execution with the `Model` tag it has authority for. In this way, it controls precisely how its sensitive data are used by an invoker from a different organization. On the other hand, the `Sales Analyzer Service` is a piece of third-party code that multiple companies entrust their sales data to. Each company can examine the code and verify that no other company can learn too much about its company's sales data by running this authority closure. Another difference is that the authority of the first closure is fixed while that of the second changes as new companies enroll in the service.

The P_{public} principal turns out to be quite useful. At first, we were going to assign different principals to each code segment but later realized that many sections can run without any authority and they are better run in a fail-safe manner with the P_{public} principal. For example, if the `Customer Management Service` had retrieved the profile for Bob rather than Alice, this programming error will be caught as the final coupon will not be able to display on the kiosk.

Although there are no file updates in this application scenario, performing updates can be tricky as the developers have to be careful in considering the order for retrieving sensitive data and calling different functions. For example, if the `Sales Analyzer Service` needs to update a log at the end of its computation (e.g., to know how to charge companies for usage of its service), it is prevented from doing so since it is called after the process became tagged with the `StoreAInfo` tag that this service is unable to declassify.

7.2 Clinical Medical Research System

Clinical information systems must comply with legal policies in protecting sensitive patient health data. Electronic Health Records (EHRs) are accessible by various physicians, medical specialists and hospital staff at primary care facilities. Standards such as the Health Insurance Portability and Accountability Act (HIPAA) [8] are put in place to address the security and privacy of health data. This application scenario examines some of the design considerations for software that may run on physicians' desktops and by medical researchers and hospital staff. Parts of this scenario have been presented in earlier chapters to demonstrate Aeolus programming constructs. Here, we elaborate with some additional examples.

7.2.1 Application Scenario Description

This scenario captures a set of applications that is commonly found in clinical settings. We focus on four programs: `Patient Registry`, `Physician's Desktop`, `Insurance Billing Generator`, and `Trend Analyzer`. Front-desk staff uses the `Patient Registry` to add new patients or update their mailing and billing addresses. The `Physician's Desktop` is intended to assist doctors during patient visits to record measurements and notes on observed symptoms, diagnosis and prognosis. This desktop software is available at various physicians offices and terminals in the hospital. A physician is required to log-in to the system with his/her username and password. Patient data are stored in a central data repository in this system. The `Insurance Billing Generator` is a background process that goes through information on patients' visits to determine the charges for the consultations and medical procedures preformed. This program translates this information into standardized medical billing codes and submits them to the patient's insurance company. The `Trend Analyzer` can be used by researchers to study disease progression and correlations that can improve decision-making for diagnosis and treatment of diseases.

7.2.2 Data Security Concerns and IFC Program Structure

Patient data are collected, handled and stored by the healthcare facility. These healthcare providers are required to protect such sensitive data. The HIPAA Privacy Rule categorizes many of the above (demographic information, medical record, payment history) as *Protected Health Information*. It regulates their use and requires providers to make a reasonable effort to disclose only the minimum necessary information required to achieve various clinical operations.

While physicians should have the right to read and update patient's medical record for patients that they are responsible for, the physician should not be able to view demographic information (e.g., his/her patient's home address). On the other hand, front-desk workers needs to update this information but they should not be able to disclose patient's medical history. Billing information contains a summary of procedures done at hospitals and clinics that is generated from the examination of physicians' and nurses' notes. While the payment data are less sensitive than the original medical history, this information should still be protected from staff outside of the finance department. The aggregation of numerous patient records can reveal patterns advancing the field of medicine. However, these research studies should only involve patients that have given informed consent.

Data Categorization

To limit the amount of sensitive data used by different applications, we use different tags to tag patient data. Demographic data should be handled with great care as they contain personally identifying information. The hospital administration uses a tag $T_{Demographic}$ to tag the file that stores such information for patients and staff. Logs on generated payment requests are kept on persistent storage using the tag $T_{Payment}$ to protect data secrecy. Research findings are important intellectual property and are kept in data repositories marked with $T_{Proprietary}$ sensitivity. For each patient, there are two tags: $T_{PatientSign}$ and $T_{PatientMedical}$. The first can be used to label the integrity of signed consent forms while the second is used to tag the patient's medical record.

Management of Principals

Each registered patient has a principal ID (e.g., `Alice`) as shown in Figure 7-3. Each hospital staff also has a principal ID. For example, there is a principal ID for physician Carol, one for front-desk worker Dave, one for billing staff Eve, and one for researcher Fred. There is an overall hospital administration principal `Clinic-Admin`. As staff join and leave the hospital, the administrator assigns them different roles by using act-for delegations. For example, `Dave` is set to act-for `FrontDesk Worker` and `Fred` is set to act-for `Researcher`. By acting for these principals, Dave and Fred are able to obtain the authority needed to get their job done. Through the `FrontDesk Worker`, Dave can retrieve patient's demographic data to schedule appointments. The `Researcher` principal gives Fred the ability to study the hospital's proprietary research data.

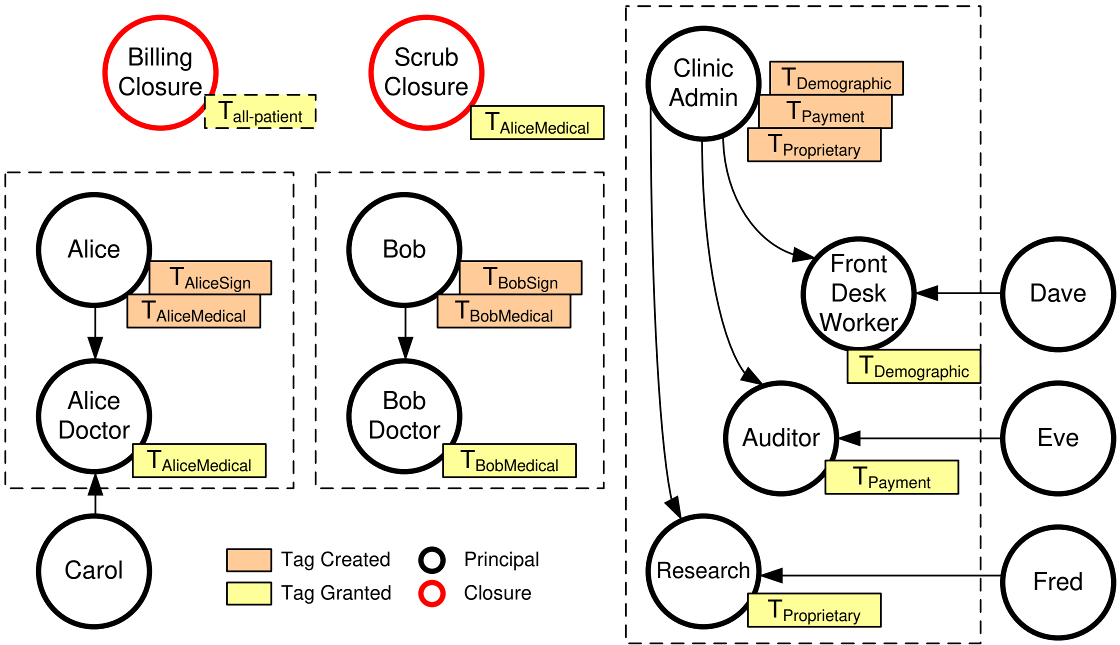

Figure 7-3: Principal Hierarchy in Clinical Application Scenario

The hospital administrator creates the tags $T_{Demographic}$, $T_{Payment}$ and $T_{Proprietary}$. It delegates the $T_{Demographic}$ tag to the `FrontDesk Worker` role since it needs to handle such information while the `Researcher` role is delegated the $T_{Proprietary}$ tag.

When a new patient enters a hospital, a new principal ID is generated for the patient along with the two patient-specific tags mentioned above. The $T_{PatientMedical}$ tag is created as a sub-tag of the top-level $T_{all-patients}$ tag. Hence, the authority for the patient's $T_{PatientMedical}$ tag is given to the `Billing Closure` which has authority for the $T_{all-patients}$ tag. The patient is asked whether he/she wants to participate in research studies and consent forms are signed and kept on file. If the patient agrees to be part of the study, the authority for the $T_{PatientMedical}$ sub-tag is given to the `Scrub Closure`. The signed consent form is stored securely and protected with the tag $T_{PatientSign}$ in its integrity label. Furthermore, the patient specifies the physicians and specialists that he/she has been referred to allowing them to act on behalf of him/her in handling medical data. In this example, Carol is Alice's doctor and `Carol` is made to act-for `AliceDoctor`.

Patient Registry

When a patient enters the hospital, Dave, the `FrontDesk Worker`, looks up the patient's address information and checks that it is still up-to-date. If the patient has moved, he enters the new information into the system. If the patient is new to the clinic, a new entry is added to the patient registry. This registry is protected by the $T_{Demographic}$ tag in both its secrecy and integrity labels. Dave is able to update the registry since he runs his desktop software with the `FrontDesk Worker` role and this role is authoritative for the $T_{Demographic}$ tag which proves that it has the required authority to write to the registry.

Insurance Billing Generator

Billing is a background process that Eve the internal auditor runs every reporting period to generate payment information. Eve uses the payment information generated by `Billing Closure`. The `Billing Closure` contains code that leaves out patient medical details and translates visit information into procedure billing codes. In this way, the `Billing Closure` is trusted with declassifying patient records and is given authority for the $T_{all-patient}$ compound tag. This compound tag covers both the $T_{AliceMedical}$ and $T_{BobMedical}$ sub-tags. To ensure that the disclosure of this payment information is controlled, this code attaches the tag $T_{Payment}$ to the resulting data before writing it to a log. Hence, Eve must run her generator software with the `Auditor` principal authoritative for the $T_{Payment}$ tag.

Trend Analyzer

The `Scrub Closure` is able to read in selected patients' data and de-sensitize them by anonymizing patient data and obfuscating fields that are not needed for research purposes. Researcher Fred uses this closure to examine scrubbed patient records for ones that have proper patient consent. In this example, Alice has given consent while Bob has not and so this authority closure has been delegated $T_{AliceMedical}$ and not $T_{BobMedical}$. Researcher Fred can study and analyze this data for example in determining the correlation between patients with high blood pressure and heart disease. Fred uses the $T_{Proprietary}$ tag to protect his research findings.

Physician's Desktop

Carol is a physician responsible for many patients. Carol has to keep records of different patients separate and avoid mistakenly updating the wrong medical records. When Alice the patient walks into Carol's office, Carol switches the `Physician's Desktop software` to run with the `AliceDoctor` prinicipal. This allows Carol to review Alice's symptoms from previous visits and enter new consultation report into the system. During this time, these computations and updates are executed within a process that has the $T_{AliceMedical}$ tag in both its secrecy and integrity labels, ensuring that it is reading Alice's data and updating only her record.

7.2.3 Programming Experience

The usage of the principal hierarchy in this Clinic example is quite different from the Retail Kiosk example. The Clinic example is a closed organization and the principal hierarchy is dominated by a `clinic-admin` principal. The hierarchy is used to define roles that different users are assigned to in this environment. As patients, doctors and hospital staff logs on to run different software, they act-for these roles and inherit the needed authority to get their work done. By defining roles, the principal hierarchy can be used as a way to manage patients and employees joining and leaving the system; it also nicely handles changing patient-doctor relationships.

The use of tags was important in representing different types of data that can belong to a single user such as a patient. Tags provide fine-grained control where authority can be limited to a very specific piece of data (e.g., $T_{Demographic}$ for demographic data and not patient records or payment information). Furthermore, compound tags provide a very convenient short-hand to grant authority for all patients' tags to the `Billing Closure`.

7.3 Secure Wiki

For our final study, we took an existing web application, ScrewTurnWiki [65], and extended it with information flow security. ScrewTurnWiki is a relatively small single server application. Here, we examine the Aeolus design from the viewpoint of modifying the wiki's code base to use our mechanisms in protecting sensitive user and system data.

7.3.1 Application Scenario Description

ScrewTurnWiki 2.0 is a popular open-source .NET wiki engine that provides features common to most wikis such as page creation and editing, simple user management, and support for third-party plug-ins. ScrewTurnWiki can be used to host private blogs as well as collaboration web sites with many users. Users create wiki pages that are either publicly visible or visible only to user account holders.

ScrewTurnWiki has a modular storage interface that can interact with file systems, databases and custom storage solutions. Moreover, it has well-defined APIs that allow third-party developers to insert their code at various stages during the processing of a page request. For example, there is a plug-in that provides support for different languages and another that tracks visits to different wiki pages. However, ScrewTurnWiki provides little support for security. This is especially of concern because of insertion of third-party code, since this means there is an expanding code base that might leak sensitive information.

7.3.2 Data Security Concerns and IFC Program Structure

We show two examples of how Aeolus can be used to strengthen data security in ScrewTurnWiki. The first example deals with sensitive user profile data such as passwords and e-mail addresses. The second example shows how to handle untrusted plug-ins such as the Basic Statistics Plugin 1.3.3 module downloadable from ScrewTurnWiki's website.

Like many web applications, ScrewTurnWiki is deployed as a server application where all modules within the application have the same authority. For example, the login code as well as the page formatter code can access files containing user passwords. Any method can wipe out or corrupt server data kept on logs and application caches. In the first example, we isolate sensitive user profile information and make use of Aeolus FS to store them persistently. The file containing sensitive user profile data is protected by the tag T_{admin} in its secrecy label and we allow only certain code such as the login module to use such data. The login module runs within an authority closure and the anonymous principal of this closure is given authority for T_{admin}.

When a new user account is created, the non-sensitive part of the profile data is stored as-is and only a small set of sensitive data (password and e-mail) is redirected to the Aeolus FS. Most of the user registration code is unmodified. When a user logs in to the wiki, an authority closure takes in the username and password and returns a boolean value on the result of the password check. This closure fetches the sensitive profile data from the Aeolus FS and declassifies the T_{admin} tag before the boolean value is returned. The checking code that runs in the authority closure consists of less than 10 lines of code. A programming error in these few lines can inadvertently disclose the password, however, a bug in the remaining 18K lines of the ScrewTurnWiki code cannot.

The second example concerns the use of the Basic Statistics plugin. Plug-in modules like these are written by open-source and commercial developers with varying levels of programming skills. They are valuable in enriching the core wiki engine but care must be taken that they do not reduce the overall security of the application.

The Basic Statistics plugin is triggered whenever the server application is about to return a wiki page to the user. It tracks all visits to wiki pages and maintains a log of such information. For each page access, it records the IP address, user browser/OS, URL, language, username (if any) and access time. From the log, various statistics such as access pattern and frequency can be derived. In most cases, the complete log is used only for administrative and auditing purposes. But nothing prevents its release and there are clear privacy issues if this should happen. Furthermore, an error in this logging code can place copies of wiki pages in the log and disclose them to users that are not intended to see them. We use the Aeolus FS to maintain the log in a file with the tag T_{stats} in its secrecy label. Most of the plug-in code runs with the P_{public} principal and only small sections of code where summarized information such as total number of accesses on a page is revealed run with more authority via an authority closure.

7.3.3 Programming Experience

We were surprised to find that with these small modifications, we were already able to strengthen ScrewTurnWiki against potential security vulnerabilities in different parts of its code base. The approach we found useful in this exercise was to isolate sensitive data, identify small code sections that require authority to disclose sensitive data, and execute all remaining code with no authority (using P_{public}).

In terms of the amount of retrofitting needed, this depended on the data structures used in the existing code. For the password protection, we simply left the data structures as-is but scrubbed out the sensitive portions. We introduced new data structures for storing the sensitive portions of a user profile and redirected code that accessed them. These changes were needed only in a handful of places and by scrubbing, we ensure that if we missed a reference (probably this would be an error in the wiki code), no security leak is possible.

The reason it was necessary to scrub the data is because a large portion of the code in the application was implemented in ASP.NET pages or web forms that are invoked by Microsoft IIS which did not run on top of Aeolus. Therefore, we could not control how information flowed in that code. By scrubbing the sensitive data, we were able to ensure that if any code in the application accessed the data structures where it used to be stored, still no leaks could occur. Figure 7-4 shows how we modified ScrewTurnWiki to redirect the execution to code that runs on Aeolus.

ScrewTurnWiki has the potential to support many plug-ins. We found that application-level tags and the ability to delegate and revoke authority to be important. New plug-ins may require a different collection of authority and obsolete plug-ins should have their authority revoked. The authority state provides a central

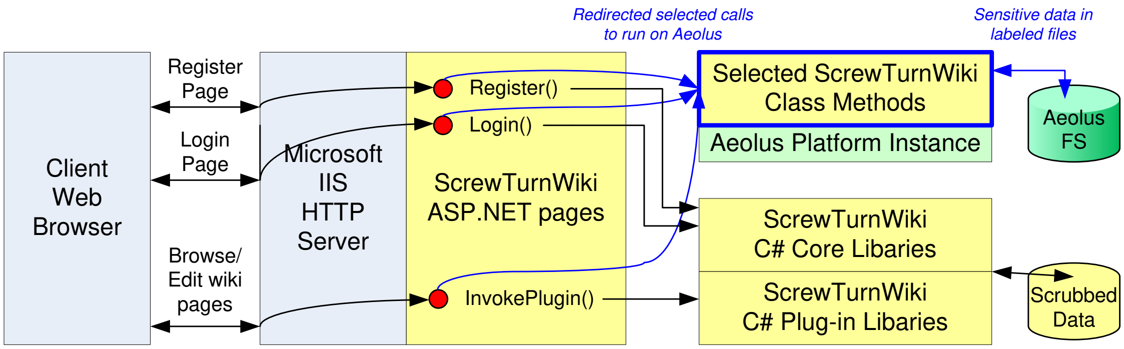

Figure 7-4: Modifications to ScrewTurnWiki

management point for these decisions.

Chapter 8

Related Work

Information flow control is not a new concept and it was first studied in the 1960s. Lampson's paper on the confinement problem [36] pointed out the insufficiency of using only access control to protect sensitive data. Information leakage can happen when a subject authorized to access an object discloses it to another subject not authorized to access it. Information flow control is a set of techniques that alleviates this problem by enforcing how data is disseminated and propagated. The Bell-LaPadula model [1] and the Biba model [2] are some of the earliest formal models that addressed the confidentiality and integrity issues in information flow. These models were designed to protect classified and secret information for government and military use and focused on a specific form known as Multi-Level Security (MLS) [58].

Operating systems dating back to the 1960s have provided some information flow isolation based on these models. At the operating system level, end-to-end information flow control can be tracked and enforced since the OS has access to all resources. ADEPT-50 [64] applied the high-watermark model to keep track of the highest security level of objects that have been opened. IX [40] and LOMAC [23] impose similar kinds of security models in the kernel based on a lattice of sensitivity labels. These systems require a centralized unit to assign users to different classification or privilege levels.

In the 1990s, Myers and Liskov [45] introduced the concept of *decentralized information flow control*. This approach makes it possible to use discretionary controls with policy decisions delegated to individual users and does not rely on one administrator configuring all policies correctly.

Aeolus and more recent systems support decentralized information flow control where different data categories can be owned by modules. Though many systems have provided some information monitoring (via processor micro-architectures [59, 53, 13], emulators [46, 4, 11], virtualization techniques [12, 67, 24, 56], security extensions [54, 22, 47], and web-browser support [62, 63, 9, 66, 5, 39]), we focus our discussion on the two lines of research that are most closely related to Aeolus: the programming language work and the operating system work.

8.1 Programming Languages

Myers and Liskov introduced decentralized information flow in JFlow [45, 43], a Java-based programming language. JFlow and its successor Jif [44] allow program to define a security policy for data, by annotating program constructs with data security labels, and uses the compiler to track and enforce information flow within a single program. They rely on a type system and static analysis, which have the advantage that they can conservatively identify information flow, providing stronger security assurances (e.g., non-interference property). Their labels are expressed in terms of principals (owners) and the security policies an owner wishes to impose on other principals. For example, each variable has a label and one can only write (i.e. declassify) it to the printer if the code is running with a principal that is allowed by all owners.

The initial language work did not provide mechanisms for interacting with OS resources such as files and sockets. Static analysis relies on closed-world assumptions with guarantees being made on a single program, and dynamic extensions can easily invalidate these guarantees [26]. The approach assumed a centralized principal hierarchy where programs can perform dynamic checks on act-for relationships. However, the work did not address updates to the principal hierarchy. A class method can be associated with a principal and in this way, they provide similar guarantees as our authority closures. These languages were intended to support a single sequential program and had no support for concurrency and for running a long-lived dynamic system.

There has been a large body of follow-on work ([30, 28, 31, 55, 7, 72, 57, 29, 6]). Here, we review the work that attempts to bridge the gap between this static way of handling information flow and real-world applications and operating environments since this is most relevant to Aeolus.

Jif guarantees *non-interference* within a single application but does not provide support for files and sockets, which are needed in real-world applications. SIESTA (Service for Inspecting and Executing Security-Typed Applications) [30] is an operating system service that combines Jif and SELinux [47] to allow labeling to extend from files and sockets to applications. It allows the operating system to use its mandatory access control (MAC) module to pass labeled data into an application (via an API for Jif) and it ensures that the application complies with the OS MAC policy (via compliance analysis). It does this by using SELinux to provide the MAC functionality.

Hicks et al [28] showed via the development of the JPmail e-mail application how Jif can be extended for designing real-world applications. They showed how to build a "principal store" that uses public-key infrastructure to provide uniqueness and persistence.

When the principal hierarchy changes, this can violate information flow assumptions of running programs. In particular, revocations can invalidate the authority of a process if it has used authority derived from its actee. Hicks et al [31] permitted dynamic updates to the principal hierarchy by placing restrictions on updates, disallowing ones that will violate properties of already executing programs. Swamy et al [55] introduced transactional semantics on principal hierarchy updates and allowed them to be made by contaminated programs by adding labels to the PH.

SIF [7] is a framework for building web applications using Jif. It addresses the limitations of security typed languages for reasoning about security in a dynamic external environment. In particular, it addresses authentication and session-based behavior of web applications through *authorization closures* and *session principals*. *Authorization closures* are used to obtain the authority of application users after they have authenticated themselves (e.g., with a password). They provide a translation of "user principals" to Jif's principals. A *session principal* is then assigned to act on behalf of the now logged-in user. This work shows how a Jif web application can interact with an untrusted client browser to provide end-to-end guarantees.

There has also been a lot of work in extending the type system to provide provable guarantees for dynamic mechanisms such as dynamic security labels [72] and dynamic principals [57] to support labels and principals that are not known at compile time while still preserving the non-interference property. Dynamic security labels allow the type-system to potentially support dynamic checks at the reading and writing of files and database records. Dynamic principals integrate static checks with external notions of principals (e.g., users authenticated through a public key infrastructure). An extension to Jif has been proposed in [29] to support *declassifier* functions using a relaxed model called *non-interference modulo trusted methods. Robust declassification*, defined in [6], ensures that an entity that can influence the behavior of a system (for example, by providing or modifying data or code) is unable to observe more information than an entity that cannot influence system behavior. The key observation for enforcing robustness is to ensure that if a declassification reveals information to attacker `A`, then `A` is unable to influence either the decision to declassify or the data to be declassified. Jif was extended with a simple dependency check at `Declassify` to provide this property.

8.2 Operating Systems

The operating systems work ([18, 60, 69, 71, 68, 35, 34, 50]) differs from the programming language work in a number of important ways.

First, rather than expressing policies using principals, these systems use tags. Tags identify compartments and provide a way of grouping related items, i.e., ones that are all managed using the same security policy.

Second, they use dynamic rather than static checking. As a process runs the system checks its information flow status. Dynamic checking is completely accurate:

a process's labels exactly reflect what it did. This differs from the static approach, which must sometimes err in a safe direction, so that the labels are more restrictive than necessary. Additionally, with static checking, a fine-grained analysis is possible so that different variables can have different labels. This isn't possible with a dynamic approach although our boxes provide support for doing this in a limited way.

Third, the operating system work is based on capabilities, which are passed among processes and can be stored for later use. They have capabilities for tags and in this way, they are able to avoid having principals in the model, although they do have to allow for a way for users to log on and obtain the capabilities they used in the past. By contrast, the programming language work and Aeolus assume the existence of authority state, which is consulted to determine whether security sensitive actions can be allowed.

A final point is that the operating systems work uses labels both for access control and information flow control. Aeolus and most of the programming language work keep them separate.

Asbestos

Asbestos [18, 60] is the first operating system with decentralized information flow control. All the operating systems work has followed this model.

Asbestos attaches labels to operating system processes and provides a UNIX-based operating system. Labels are used both to reflect a process' contamination and to express the process' authority as a set of capabilities. Their labels are expressed in terms of tags and levels, without any use of principals. Asbestos provides a way to grant capabilities to another process. Revocation is difficult in a capability-based system and is not addressed in Asbestos.

Asbestos uses standard capability-based mechanisms to provide system-wide persistence of authority. A pickle is a special file that stores privilege for a single tag. A process can create a pickle file that can later be un-pickled to retrieve the authority.

HiStar

HiStar [69, 68] provides information flow at the level of an operating system kernel; this way it is able to have a much smaller secure base than Asbestos. The kernel is intended to be used to build an operating system and the utility of the kernel was demonstrated by implementing an untrusted user-level UNIX emulation layer. HiStar is not intended to support user code directly.

HiStar labels are similar to those of Asbestos. The work improved on Asbestos (as described in the original paper [18]) by requiring threads to explicitly request safe label changes; this is important because it avoids covert channels possible in Asbestos via IPC.

HiStar uses capabilities and has a single-level store to maintain system-wide persistence. This single-level store is used to maintain capabilities given to threads and gates as well as to implement a user-level file system out of containers and segments. Operations are performed on memory-mapped files in the thread's address space.

HiStar supports IPCs via *gates*. Gates provide a protected control transfer and make resource allocations explicit to avoid covert channels when invoking methods. They bind privileges with a well-defined entry point for the callee' process to execute the method (e.g., this gate is for running `callee.Foo()` with authority for T_{Alice}). In this way, gates are similar to authority closures, however, they are too low-level for writing user applications. HiStar maintains labels for threads and threads can access shared address spaces. Address spaces are page-aligned and hence, HiStar provides for page-level data sharing.

DStar

DStar [71] extends HiStar over the network, carefully controls the covert channels that may exist in resource allocation, and delegates trust using delegation certificates. One of DStar's design goals is decentralized trust: there is no trusted authority like Verisign to provide reliable naming of machines; instead each machine is identified by a public-private key pair. Category names are *self-certifying* and identify the machine that the category belongs to. DStar maintains the private-public key information in local *exporters* and uses *address certificates* to distribute this ⟨IP address, key⟩ binding. *Delegation certificates* are signed by the owner machine to delegate a category to a receiving node. DStar relies on renewable leases for revocation.

Flume

Flume [35] is a reworking of Asbestos, with several important differences. First, Flume is implemented by a user-level reference monitor rather than a DIFC operating system. This approach avoids the need to modify the operating system; additionally, it means that Flume is unaffected by new release of operating system code. However, compared to Asbestos, Flume has a larger trusted base (the reference monitor plus the operating system).

A second difference is that Flume provides much simpler labels than Asbestos and HiStar because it breaks the single label of these systems into their three components (a secrecy label, an integrity label, and an ownership label that keeps track of read and write capabilities).

Flume keeps track of capabilities using a central *tag registry* that maps login tokens to capabilities. Flume's *setlabel* binds a login token (and hence capabilities) with a program. In this way, setlabels are similar to authority closures; however, they are implemented as special files that hide the login tokens with secrecy and integrity labels that limit who can invoke it, whereas authority closures can be invoked by anyone since their security is guaranteed by analyzing the code.

A final point is that Flume provides endpoints as a way to reduce the cost of label checking as messages are received or files are used. These endpoints were an inspiration for the way we implement file-streams in Aeolus.

Laminar

Laminar [50] is based on Flume. Like Flume, it is implemented on top of the OS, but because Laminar includes some programming language as well as kernel extensions, its trusted base includes in addition to the operating system, a modified Java virtual machine. Laminar introduces support for processes to shared objects, an ability that is missing in all of the OS work except for HiStar, where sharing happened at the level of pages. Sharing happens through a linguistic mechanism called a *security region*; programs using Laminar must use the extended language that supports regions. A security region is a lexical scope within a process that can have its own labels. Laminar attaches Flume labels to each region and checks when data access crosses two regions. However, these checks can be costly.

Chapter 9

Conclusions

9.1 Methodology

This thesis has presented Aeolus, a new distributed platform intended to support the development and deployment of applications that can be trusted to avoid accidental release of information. Aeolus supports applications through a new, simple security model based on information flow control. In addition, Aeolus provides a number of new features: anonymous authority closures, compound tags, boxes, and shared volatile state. These mechanisms provide needed expressive power to application developers and make it easier to develop safe applications.

Aeolus advocates a principled approach to secure software design.

- **Understandable and Easy to Use.** The programming model introduces a set of programming constructs. These constructs are intuitive and easy to understand.

- **Sufficiently Expressive.** The programming model is sufficiently expressive that a developer can specify common data flow patterns.

- **Debuggable and Auditable.** The programming model allows developers to reason about their IFC constraints and can provide interfaces for extending auditing and debugging features. The use of principals and a logically centralized AS make it possible to build audit trails.

- **Implementable.** The programming model is enforceable and we have shown an implementation of the model that does not have significant performance and scalability penalties.

9.2 Infrastructure

A prototype implementation based on C# and .NET showed that an Aeolus platform can be designed to implement our programming model. The results show that the performance is reasonable, and that the approach we take to caching authority state is effective. We have provided persistent storage, shared volatile state and process isolation that demonstrate the different components of our model.

9.3 Future Work

This thesis has demonstrated one design of the distributed computing platform. We plan to look at changing how we store the authority state. At present, this is stored in a database, which may be too heavy-weight and inefficient. We would like to investigate alternative structures, such as direct use of disk storage. Also, in our prototype, cores are variable size and hence, optimizations of core storage allocation and access are difficult, and we are looking for approaches to managing cores that better match the physical constraints of server hardware.

Auditing is an important feature that can strengthen the security of systems by deterring misuse and holding users accountable for their actions. We want to log every event with security implications, along with information about who did it, when and how. Such information can be valuable and we intend to study what types of auditing information are useful in a system like Aeolus and how our platform must be extended to capture and store this data. In addition, since principal IDs are opaque to Aeolus, auditing requires us to allow application developers to store information about real people who are associated with a particular principal ID. Such data will pose privacy issues and we may want to store them in an encrypted form. There are also questions about what types of queries on the audit trail our system can support and what restrictions must be in place to protect the privacy concerns of such data.

Another future direction is to integrate databases into the model in a flexible way that allows "multi-labeled' relations. The problem here is that we need fine-grained labels and having a single set of labels per relation is not sufficient. Consider a database table that contains patient records where tuple contains information about a particular patient; at the very least, we want a patient-specific label on each record. How can we design database systems that support this efficiently? Furthermore, what should the semantics be for queries on such relations?

Our prototype platform relies on type-safety to isolate processes and hence is limited to applications written in languages such as C# and Java. How do we go about supporting unsafe languages like C? Perhaps this will bring our platform architecture closer to systems like Flume and HiStar, which have a smaller secure base. A big issue will be how processes in app-objects and how shared state can be implemented efficiently.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Aeolus Programming API

A.1 Aeolus Basic Model

PRINCIPALS
CreatePrincipal(out P1) Creates a new principal in the process' principal's core and returns the new principal ID *P1*. Constraint(s): None. Effect(s): Process principal *P* acts for the new principal *P1*.
CreatePrincipalInNewCore(out P1) Creates a new principal in a new core and returns the new principal ID *P1*. Constraint(s): None. Effect(s): Process principal *P* acts for the new principal *P1*.
CreatePrincipal(in P2, out P1) Creates a new principal in the same core as principal *P2* and returns the new principal ID *P1*. Constraint(s): None. Effect(s): Process principal *P* acts for the new principal *P1*.

TAGS
CreateTag(out T) Creates a new top-level tag *T*. Constraint(s): None. Effect(s): Process' principal *P* has authority for tag *T*.
CreateSubTag(in T1, out T2) Creates a sub-tag of *T1* and returns the new sub-tag ID *T2*. Constraint(s): *T1* must be a top-level tag. Effect(s): Process' principal *P* has authority for tag *T2*.
CreateSubTag(in T1, in P1, out T2) Creates a sub-tag of *T1, s*tores the tag info in the same core as principal *P1* and returns the new sub-tag ID *T2*. Constraint(s): *T1* must be a top-level tag. Effect(s): Process' principal *P* has authority for tag *T2*.

DELEGATIONS AND REVOCATIONS
ActFor(in P1, in P2) Adds an act-for link from actor principal *P2* to actee principal *P1*. Constraint(s): Process' principal must act for principal *P1*. Effect(s): Principal *P2* has authority for all tags that principal *P1* has authority for.
RevokeActFor(in P1, in P2) Removes the act-for link from principal *P2* to principal *P1*. Constraint(s): Process' principal must act for principal *P1*. Effect(s): Principal *P2* no longer has any derived authority from principal *P1*.
Delegate(in T, in P1, in P2) Gives authority for tag *T* from grantor principal *P1* to grantee principal *P2*. Constraint(s): Process' principal must act for the grantor principal *P1*. The grantor principal *P1* must have authority for tag *T*. Effect(s): Principal *P2* has authority for tag *T*.
RevokeDelegate(in T, in P1, in P2) Revokes authority for tag *T* from grantor principal *P1* to grantee principal *P2*. Constraint(s): Process' principal must act for the grantor principal *P1* and the tag *T* must have been previously delegated from *P1* to *P2*. Effect(s): Principal *P2* no longer has authority for tag *T* via principal *P1*. All transitive delegations are also revoked.

Label Manipulations
AddSecrecy(in T) Adds tag *T* to the process' secrecy label. Constraint(s): None. Effect(s): Process' secrecy label includes tag *T*.
RemoveIntegrity(in T) Removes tag *T* from the process' integrity label. Constraint(s): None. Effect(s): Process' integrity label does not include tag *T*.
Declassify(in T) Removes tag *T* from the process' secrecy label. Constraint(s): Process' principal must have authority for *T*. Effect(s): Process' secrecy label does not include tag *T*.
Endorse(in T) Adds tag *T* to the process' integrity label. Constraint(s): Process' principal must have authority for *T*. Effect(s): Process' integrity label includes tag *T*.

A.2 Aeolus Extensions

AUTHORITY CLOSURE
CreateClosure(in key, out CL1) Creates a new closure with the *key* and returns the closure ID *CL1*. Constraint(s): None. Effect(s): A new anonymous principal *PA* is generated and bound to this key and closure ID. This principal has no authority.
CreateClosure(in key, in P1, out CL1) Creates a new closure with the *key* , stores the closure info in the same core as principal *P1* and returns the closure ID *CL1*. Constraint(s): None. Effect(s): A new anonymous principal *PA* is generated and bound to this key and closure ID. This principal has no authority.
ClosureActFor(in P1, in CL1) Adds an act-for link from the anonymous principal *PA* of closure *CL1* to principal *P1*. Constraint(s): Closure *CL1* must exist. Process' principal must act for principal *P1*. Effect(s): Anonymous Principal *PA* is authoritative for all tags that principal *P1* has authority for.
ClosureDelegate(in T, in P1, in CL1) Gives authority for tag *T* from grantor principal *P1* to grantee anonymous principal *PA* of closure *CL1*. Constraint(s): Closure *CL1* must exist. Process' principal must act for the grantor principal *P1*. The grantor principal *P1* must have authority for tag *T*. Effect(s): Anonymous Principal *PA* has authority for tag *T*.

BOXES
CreateBox(in outerS, in outerI, in innerS, in innerI, out AeolusBox b) Creates a new empty box *b* with the specified labels. Constraint(s): The *innerS* and *innerI* labels must be at least as restrictive as the *outerS* and *outerI* labels. The process labels must allow the write based on the outer labels. Effect(s): Box *b* exists in the process' heap.
b.GetInnerS() Retrieves the inner secrecy label of box *b*. Constraint(s): Box *b* must exists. The process labels must allow the read using *b*'s outer labels. Effect(s): None.
b.GetInnerI() Retrieves the inner integrity label of box *b*. Constraint(s): Box *b* must exists. The process labels must allow the read using *b*'s outer labels. Effect(s): None.
b.GetContents(out content) Retrieves the *content* of box *b*. Constraint(s): Box *b* must exists. The process labels must allow the read using *b*'s inner labels. Effect(s): None.
b.PutContents(in content) Copies *content* into box *b*. Constraint(s): Box *b* must exists. The process labels must allow the write using *b*'s inner labels. Effect(s): None.

SHARED STATE
SHARED OBJECTS
CreateObject(in o, out s) Creates a new shared state object that is a copy of object *o* in shared state and returns a new SharedObjectID *s*. Constraint(s): None. Effect(s): Object *o* exists in shared state with the same labels as the process'.
CreateObject(in o, in S, in I, out s) Creates a new shared state object that is a copy of object *o* in shared state with secrecy label *S* and integrity label *I* and returns a new SharedObjectID *s*. Constraint(s): Process labels must be no more restrictive these labels. Effect(s): Object *o* exists in shared state.
GetObject(in s, out o) Retrieves a copy of shared object *o* identified by SharedObjectID *s*. Constraint(s): Shared object must exist at ID *s* and the process labels must allow the read using *o*'s labels. Effect(s): Object *o* exists in the requestor process' heap.
ReplaceObject(in s, in o) Replaces the current object associated with SharedStateObjectID *s* with a copy of object *o*. Constraint(s): Shared object must exist at ID *s* and the process labels must allow the write using *o*'s labels. Effect(s): Shared object associated with ID *s* contains object *o*.
ReplaceObject(in s, in o, in listS, in listI) Replaces the current object associated with SharedStateObjectID *s* with a copy of object *o*. Constraint(s): Shared object must exist at ID *s* and the process labels adjusted with declassification of tags in *listS* and endorsement of tags in *listI* must allow the write using *o*'s labels. Process principal *P* must have authority for all tags in *listS* and *listI*. Effect(s): Shared object associated with ID *s* contains object *o*.
DeleteObject(in s) Removes the shared object associated with SharedObjectID *s*. Constraint(s): Shared object must exist at ID *s* and the process must have a null secrecy label and the same integrity label as the object's. Effect(s): Shared object associated with *s* is removed.

SHARED STATE
SHARED QUEUES
CreateQueue(out q) Creates a new and empty shared queue in shared state and returns its SharedQueueID *q*. Constraint(s): None. Effect(s): Shared queue *q* exists in shared state with the same labels as the process'.
CreateQueue(in S, in I, out q) Creates a new and empty shared queue in shared state with secrecy label *S* and integrity label *I* and returns its SharedQueueID *q*. Constraint(s): Process labels must be no more restrictive than these labels. Effect(s): Shared queue *q* exists in shared state.
Enqueue(in q, in o) Appends object *o* at the end of the shared queue with SharedQueueID *q*. Constraint(s): Shared queue must exist at ID *q* and the process labels must allow the write using *q*'s labels. Effect(s): Object *o* is appended to the shared queue.
Enqueue(in q, in o, in listS, in listI) Appends object *o* at the end of the shared queue with SharedQueueID *q*. Constraint(s): Shared queue must exist at ID *q* and the process labels adjusted with declassification of tags in *listS* and endorsement of tags in *listI* must allow the write using *q*'s labels. Process principal *P* must have authority for tags in *listS* and *listI*. Effect(s): Object *o* is appended to the shared queue.
GetQueue(in q, out List[o]) Retrieves the list of objects *List[o]* in shared queue identified by SharedQueueID *q* and clears the queue. Constraint(s): Shared object must exist at ID *s* and the process labels must be the same as *q*'s labels. Effect(s): Shared queue is empty.
WaitAndDequeue(in q, out o) Blocks until the shared queue associated with SharedQueueID *q* is non-empty, returns and removes the first object *o* in this queue. Constraint(s): Shared object must exist at ID *s* and the process must be the same as *q*'s. Effect(s): Removes the first entry of the queue.
DeleteQueue(in q) Removes the shared queue associated with SharedQueueID *s*. Constraint(s): Shared queue must exist at ID *s* and the process must have a null secrecy label and the same integrity label as the queue's. Effect(s): Shared queue associated with *q* is removed.

SHARED STATE
SHARED LOCKS
CreateLock(out k) Creates a new shared state lock and returns a new SharedLockID *k*. Constraint(s): None. Effect(s): Lock *k* exists in shared state with the same labels as the process'.
CreateLock(in o, in S, in I, out s) Creates a new shared state lock in shared state with secrecy label *S* and integrity label *I* and returns a new SharedLockID *k*. Constraint(s): Process labels must be no more restrictive these labels. Effect(s): Lock *k* exists in shared state.
Lock(in k) Attempts to acquire lock on *k*. If locked, blocks until unlocked and acquires lock. Constraint(s): Shared lock must exist at ID *k* and the process labels must allow the read and write using *k*'s labels. Effect(s): Upon return, lock is acquired on shared lock *k*.
Unlock(in k) Unlocks the shared lock *k*. Constraint(s): Process labels must allow the write using *k*'s labels. Effect(s): Shared lock *k* is unlocked.
DeleteLock(in k) Removes the shared lock associated with SharedLockID *k*. Constraint(s): Shared lock must exist at ID *k* and the process must have a null secrecy label and the same integrity label as the lock's. Effect(s): Shared lock associated with *k* is removed.

A.3 Aeolus Execution

EXECUTION

abstract **AeolusCallable**
`{`
`  void Invoke();`
`}`

Fork(in C) Invokes AeolusCallable code object *C* in a new process with requestor process' principal and labels. Constraint(s): None. Effect(s): Code object *C* is copied to and invoked in the new process.
Fork(in C, in P1) Invokes AeolusCallable code object *C* in a new process with principal *P1* and requestor process' labels Constraint(s): Requestor process' principal must act for principal *P1*. Effect(s): Code object *C* is copied to and invoked in the new process.
Fork(in C, in P1, in listS, in listI) Invokes AeolusCallable code *C* in a new process with principal *P1* and requestor process' labels adjusted with declassification of tags in *listS* and endorsement of tags in *listI*. Constraint(s): Requestor process' principal *P* must act for principal *P1* and *P* must have authority for tags in *listS* and *listI*. Effect(s): Code object *C* is copied to and invoked in the new process.
Call(in C, in P1) Invokes AeolusCallable code *C* with principal *P1*. Constraint(s): Requestor process' principal *P* must act for principal *P1*. Effect(s): Code object *C* is invoked in the same process. At the end of the call, the process' principal is restored to *P* and the process' labels reflect contamination from executing call.
Call(in C, in P1, in listS, in listI) Invokes AeolusCallable code *C* with principal *P1* and requestor process' labels adjusted with declassification of tags in *listS* and endorsement of tags in *listI*. Constraint(s): Requestor process' principal *P* must act for principal *P1* and *P* must have authority for tags in *listS* and *listI*. Effect(s): Code object *C* is invoked in the same process. At the end of the call, the process' principal is restored to *P* and the process' labels reflect contamination from executing call.

AUTHORITY CLOSURE

abstract **AeolusClosureCallable**
`{`
`  GetClosureID(out ClosureID CL1);`
`  GetCertificate(out string certifier);`
`  void Invoke();`
`}`

CallClosure(in CL) Invokes AeolusClosureCallable code object *C* with anonymous principal *PA* associated with closure ID *CL1*. Constraint(s): Code object *C* must be verified using closure *key* associated with closure ID *CL1*. Effect(s): Code object *C* is invoked in a new process. At the end of the call, the process' principal is restored to the requestor's and the process' labels reflect contamination from executing call.
CallClosure(in CL, in listS, in listI) Invokes AeolusClosureCallable code object *C* with anonymous principal *PA* associated with closure ID *CL1* and requestor process' labels adjusted with declassification of tags in *listS* and endorsement of tags in *listI*. Constraint(s): Code object *C* must be verified using closure *key* associated with closure ID *CL1* and requestor process' principal *P* must have authority for tags in *listS* and *listI*. Effect(s): Code object *C* is invoked in a new process. At the end of the call, the process' principal is restored to the requestor's and the process' labels reflect contamination from executing call.

A.4 Aeolus File System

SETUP

CreateFilesystem()

Creates a new filesystem.
Constraint(s): None.
Effect(s): The root entry of the filesystem will have the same labels as the process'.

CreateFilesystem(in S, in I)

Creates a new filesystem with root entry protected by secrecy label *S* and integrity label *I*.
Constraint(s): The process labels must be no more restrictive than these labels.

RemoveFilesystem()

Removes the filesystem.
Constraint(s): The process labels must allow the write of the root entry.

ADMINISTRATIVE OPERATIONS

CreateDir(in F)

Creates a directory with filepath *F*.
Constraint(s): The process labels must allow the read and write of the parent directory.
Effect(s): The directory will have the same labels as the process'.

CreateDir(in F, in S, in I)

Creates a directory with filepath *F*, secrecy label *S* and integrity label *I*.
Constraint(s): The process labels must allow the read and write of the parent directory. The process labels must be no more restrictive than the *S* and *I* labels.

CreateFile(in F)

Creates a file with full filename *F*.
Constraint(s): The process labels must allow the read and write of the parent directory.
Effect(s): The file will have the same labels as the process'.

CreateFile(in F, in S, in I)

Creates a file with full filename *F*, secrecy label *S* and integrity label *I*.
Constraint(s): The process labels must allow the read and write of the parent directory.

RemoveDir(in F)

Deletes the directory with filepath *F* and any sub-directories or files.
Constraint(s): The process labels must allow the read and write of the parent directory.

RemoveFile(in F)

Deletes the file with full filename *F*.
Constraint(s): The process labels must allow the read and write of the parent directory.

ListDir(in F, out listF)

Lists the files and directories in directory with filepath *F*.
Constraint(s): The process labels must allow the read of the parent directory.

READ AND WRITE ENTIRE FILE
ReadFile(in F, out buffer) Reads file with full filename *F* into *buffer*. Constraint(s): The process labels must allow the read of the file. Effect(s): Buffer contains content of file *F*.
WriteFile(in F, in buffer, in listS, in listI) Writes the content of *buffer* to file with full filename *F* Constraint(s): The process labels must allow the write of the file. In addition, the process' principal must be authoritative for the tags in *listS* and *listI*. Effect(s): File *F* overwritten with content of *buffer*.

FILESTREAMS
OpenFilestream(in F, in M, in listS, in listI, out fs) Creates a filestream for file with full filename *F* and specifies the access mode *M*. There are three access modes: READ-ONLY, WRITE-ONLY and READWRITE. Constraint(s): Process' principal must have authority for the tags in *listS* and *listI*. Effect(s): For READWRITE and WRITE-ONLY, selectively declassify tags in *listS* and endorse tags in *listI* during the write.
fs.CloseFilestream() Closes filestream *fs*. Constraint(s): Filestream *fs* must exist. Effect(s): Filestream *fs* is closed if opened.
MODE: READ-ONLY, READWRITE
fs.Read(in N, out buffer) Reads *N* bytes from the filestream into *buffer*. Constraint(s): The filestream must be opened in a readable mode. The file and process labels must allow the read of the file.
MODE: READ-WRITE, WRITE-ONLY
fs.Write(in buffer) Writes the content of *buffer* to the filestream. Constraint(s): The filestream must be opened in a writable mode. The file and process labels must allow the write of the file.

Bibliography

[1] D. Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. MITRE Technical Report 2547, Volume I, 1973.

[2] K. J. Biba. Integrity Considerations for Secure Computer Systems. MITRE Technical Report 3153, 1977.

[3] Don Box and Ted Pattison. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[6] Stephen Chong and Andrew C. Myers. Decentralized Robustness. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 242–256, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of USENIX Security Symposium 2007*, 2007.

[8] U.S. Congress. The Health Insurance Portability and Accountability Act of 1996 (HIPAA) Privacy Rule. http://www.hhs.gov/ocr/privacy/index.html, 1996.

[9] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A Safety-Oriented Platform for Web Applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society.

[10] J. Crampton and H. Khambhammettu. Delegation in role-based access control. In *Proceedings of 11th European Symposium on Research in Computer Security*, pages 174–191, 2006.

[11] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[12] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, New York, NY, USA, 2007. ACM.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 482–493, New York, NY, USA, 2007. ACM.

[14] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[15] Dorothy Elizabeth Robling Denning. *Cryptography and data security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[16] P. J. Denning. The Working Set Model for Program Behavior. In *Communications of the ACM*, 1968.

[17] Mike Downen. CLR Inside Out: Using Strong Name Signatures. MSDN Magazine, July 2006.

[18] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[19] Erick Schonfeld. Financial Exposure: Rudder inadvertently shows users each other's bank account info. TechCrunch: http://www.techcrunch.com/2009/05/19/financial-exposure-rudder-inadvertently-shows-users-each-others-bank-account-info/, May 2009.

[20] D.F. Ferraiolo and D.R. Kuhn. Role Based Access Control. In *15th National Computer Security Conference*, 1992.

[21] Financial Week. Data breech at WellPoint puts 130,000 customers at risk. http://www.financialweek.com/apps/pbcs.dll/article?AID=/20080410/REG/756233065/1036, April 2008.

[22] FreeBSD Foundation. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. *http://www.trustedbsd.org/sebsd.html*, 2004.

[23] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[24] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[25] Ian Grant. Cloud-based document storage added to iPhone. ComputerWeekly, December 2008.

[26] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, Dynamic Information Flow for Virtual Machines. In *2nd International Workshop on Programming Language Interference and Dependence*, 2005.

[27] James A. Hall and Stephen L. Liedtka. The Sarbanes-Oxley Act: implications for large-scale IT outsourcing. *Commun. ACM*, 50(3):95–100, 2007.

[28] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 153–164, Dec. 2006.

[29] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM.

[30] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick Drew McDaniel. From Trusted to Secure: Building and Executing Applications That Enforce System Security. In *USENIX Annual Technical Conference*, pages 205–218, 2007.

[31] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic Updating of Information-Flow Policies. In *Proceedings of the International Workshop on Foundations of Computer Security (FCS)*, June 2005.

[32] Josh Lowensohn. Rudder steers personal finance to your in-box. CNET News: http://news.cnet.com/8301-17939_109-10040526-2.html, September 2008.

[33] James B. D. Joshi and Elisa Bertino. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 81–90, New York, NY, USA, 2006. ACM.

[34] Maxwell Krohn. Information Flow Control for Secure Web Sites. PhD Thesis, Massachusetts Institute of Technology, 2008.

[35] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS*

symposium on Operating systems principles, pages 321–334, New York, NY, USA, 2007. ACM.

[36] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[37] Jun Li. HP Retail Kiosk Platform. http://opra.hpl.hp.com/RSA/start.html, 2008.

[38] Jun Li, Ismail Ari, Jhilmil Jain, Alan Karp, and Mohamed Dekhil. Mobile In-Store Personalized Services. In *Proceedings of 7th IEEE International Conference on Web Services (ICWS 2009)*, July 2009.

[39] Benjamin Livshits and Úlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 95–104, New York, NY, USA, 2007. ACM.

[40] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Softw. Pract. Exper.*, 22(8):673–694, 1992.

[41] Rebecca T. Mercuri. The HIPAA-potamus in health care data security. *Commun. ACM*, 47(7):25–28, 2004.

[42] Jean Moreau, Roberto Chinnici, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, W3C, March 2006.

[43] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[44] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. *http://www.cs.cornell.edu/jif*, 2001.

[45] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA, 1997. ACM.

[46] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS*, 2005.

[47] NSA. Security-enhanced Linux. *http://www.nsa.gov/selinux*, 2000.

[48] Privacy Rights Clearinghouse. A Chronology of Data Breeches. http://www.privacyrights.org/ar/ChronDataBreaches.htm.

[49] Rafael Ruffolo. Ryerson privacy breach highlights immature IT, analyst says. IT World Canada: http://www.itworldcanada.com/a/Leadership/cbee770a-74c7-44b4-aa95-9dccf1bc037b.html, February 2009.

[50] Indrajit Roy, Donald E Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[51] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-Flow Security. In *IEEE Journal on Selected Areas in Communications*, 2003.

[52] Jerry Saltzer and Mike Schroeder. The protection of information in computer systems. Proceedings of the IEEE, September 1975.

[53] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[54] SunMicrosystems. Trusted Solaris Operating System: A Technical Overview. *http://www.sun.com/software/whitepapers/wp-ts8/ts8-wp.pdf*, 2000.

[55] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing Policy Updates in Security-Typed Languages. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 202–216. IEEE Computer Society, 2006.

[56] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 20–20, Berkeley, CA, USA, 2006. USENIX Association.

[57] S. Tse and S. Zdancewic. Run-time principals in information-type systems. In *IEEE Symposium on Security and Privacy*, May 2004.

[58] U.S. Department of Defense Computer Security Center. Trusted Computer System Evaluation Criteria (The Orange Book). DoD 5200.28-STD, December 1985.

[59] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[61] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[62] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2007. ACM.

[63] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. Msr-tr-2009-16, Microsoft Research, 2009.

[64] C Weissman. Security controls in the ADEPT-50 time-sharing system. *Proc. AFIPS 1969 FJCC*, 23:119–133, 1969.

[65] ScrewTurn Wiki. ScrewTurn Wiki 2.0. http://www.screwturn.eu/, June 2009.

[66] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 233–246, New York, NY, USA, 2009. ACM.

[67] Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh. A feather-weight virtual machine for windows applications. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 24–34, New York, NY, USA, 2006. ACM.

[68] Nickolai Zeldovich. Securing Untrustworthy Software Using Information Flow Control. PhD Thesis, Stanford University, 2007.

[69] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[70] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In *NSDI 2008*, 2008.

[71] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[72] L. Zheng and A.C. Myers. Dynamic security labels and non-interference. In *Proceedings of the 2nd Workshop on Formal Aspects in Security Trust*, August 2004.