Acetone: A System Call Interface for Asbestos Labels

by

Clifford A. Frey

Bachelor of Science in Computer Science and Engineering,
Massachusetts Institute of Technology (2004)

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

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2005

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Acetone: A System Call Interface for Asbestos Labels

by

Clifford A. Frey

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2005, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Acetone is a secure operating system kernel that uses a shared address space and supports Asbestos labels. Acetone uses Asbestos labels to enable a wide variety of security policies including ones that prevent untrusted applications from being able to disclose private data. All threads run in the same address space, but have different memory access privileges. Acetone uses standard memory protection mechanisms to ensure that all memory accesses are consistent with label rules. The performance results show that these checks have a relatively low cost.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Acknowledgments

I would like to thank David Ziegler, Maxwell Krohn, Russ Cox, and Frans Kaashoek for valuable guidance and discussion. This thesis would not have been possible without them.

I would also like to thank Betsy, Laura, and Tina for their love, support, and encouragement.

This research was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425.

Parts of this thesis are adapted from and contain text written in collaboration with Petros Efstathopoulos, Frans Kaashoek, Eddie Kohler, Max Krohn, David Mazières, Robert Morris, Steve VanDeBogart, and David Ziegler.

Parts of this thesis are also adapted from

> Max Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, June 2005.

Contents

Chapter 1

Introduction

This master thesis presents Acetone, a new operating system that makes it easier to implement secure applications. Acetone combines Asbestos labels, a unified address space, and the ability to easily create new security domains.

Acetone is a redesign of Asbestos, an operating system that introduced Asbestos labels. Asbestos labels allow security domains to be easily defined in terms of label primitives. These domains can be defined in terms of both mandatory access control and discretionary access control policies. Acetone simplifies the Asbestos system call interface. The benefits from this simplification include making it easier to reason about security policies, an easier to reason about implementation, and simplifying sharing of resources between security domains.

1.1 Motivation

Current computer systems do not provide adequate security for their users. Viruses, spyware, and adware plague many users' systems, and large-scale commercial servers frequently end up disclosing sensitive information.

These problems mostly stem from two main sources. The first is that software contains bugs that allow exploits. The second is that in most personal computing environments, users are willing to run untrusted code that appears to serve a legitimate purpose.

Although it is always possible to improve the quality of application code, it is difficult to remove all security vulnerabilities. Similarly, it is possible to educate users to make them less trusting, but there are times when it is still desirable to run untrusted code. Given these difficulties, the best approach to improving security of applications is to provide better operating system support for isolating modules from one another, and to allow a user to run untrusted code with a restricted set of privileges.

The principle of least privilege states that each bit of code that executes on a machine should run with the least amount of privilege possible [17]. This secures applications by preventing a bug in one part of an application from being able to use all of that application's privileges. This principle can also help users by allowing them to run entire applications with the fewest privileges possible, therefore limiting the effect that these applications can have on their system.

1.2 Operating System Security Overview

In order to enforce modularity between different components, operating systems allow code to be run in different *domains*. A domain is a set of resources that can all be accessed by the current thread at the same time. Ultimately, in order to implement the principle of least privilege, it is important that applications can easily create many domains that each have the smallest amount of privilege necessary. It must also be simple for these domains to communicate with each other so that each domain can be small, allowing many domains to exist, each with just the privileges that it needs. By carefully analyzing how different security domains are created and used, and by looking at how data can be moved from one domain to another, it is possible to evaluate an operating system and to determine how easily it enables an application writer to use the principle of least privilege.

Different operating systems have varying interfaces for creating a domain and controlling what resources are available in each domain. For instance, most UNIX systems essentially have one domain for each process running on the system. The

memory and register state of different processes are protected from other processes. Each process domain includes access to the register state of the process, memory mapped to the process, and access to all of the file and network resources of the user who owns the process.

1.3 Asbestos Overview

Asbestos allows application writers to reach towards the principle of least privilege by giving them a set of primitives that provide easy modularization. Asbestos improves upon the UNIX security model in a number of important ways. The primary improvement is the use of labels for access control instead of basic user IDs and default process rights. Also, Asbestos allows multiple domains per process, which can only be entered or accessed in response to receiving messages from other processes.

The basic way to create a new isolated security domain in Asbestos is to create a new process. Each process has an associated security label that defines a domain containing all of the resources that the process has access to based on its label and the label rules. In order to reduce the overhead of having one process in the system for each security domain, Asbestos also supports having multiple subdomains within one process. This can be done by assigning labels to ranges of the virtual address space. These ranges can then only be accessed when the process receives a message with the appropriate label. For instance, a virtual address space range can be set up for each user who is logged in to a server, and even if there are bugs in the server, it is impossible for a user to access another user's data.

Both Asbestos processes and individual pages of memory can have security labels. It is possible for data to move from one security domain to another by sending a message from one process to another. As messages are transmitted in the system, Asbestos also keeps track of information flow by updating security labels appropriately.

A simplified SSHD implementation in Asbestos is shown in Figure 1-1. In most cases, different parts of the functionality are isolated from one another by using dif-

ferent processes. It is possible for many domains to exist in the SSHD/Protocol process because the SSHD/Demux process can send specially marked messages that give additional privileges but also restrict the SSHD/Protocol process to a sub-domain.

As an example of how an application can be broken up into different domains in Asbestos, consider a simple implementation of *SSHD* that permits network logins to the local machine, but where local users are never allowed to see each other's data. This application needs to have many privileges on the system, including the privilege to act on behalf of any user of the system. However, in order to use the principle of least privilege, most parts of the system should be running with only a few privileges. For instance, once a user has logged in, SSHD has three main modules that handle the connection. First, the *demultiplexer* maps individual network streams to specific user sessions. Second, the *protocol manager* handles all ssh-specific protocol tasks, including encryption. Last, the specific user's *shell* provides a basic interface to the users applications and data, and allows the user to perform any task allowed by that user's privileges.

In this example, in order to follow the principle of least privilege, the application should be broken up into at least a few different domains. The domain where the protocol manager executes should only contain the privilege to access the specific user's network connection and the input and output of the user's shell. The user's shell should have a domain that has full access to all of the user's privileges. The demultiplexer needs to have a domain that contains access to the network and all of the protocol managers. The demultiplexer domain must also run in a domain that has the privilege to declassify user-private data, allowing it to be sent over the network.

1.4 Problems

One problem with Asbestos is that it is cumbersome to share data with other domains, as the only way to communicate information between domains is through messages. This is a problem because it is often easier to write applications that use a

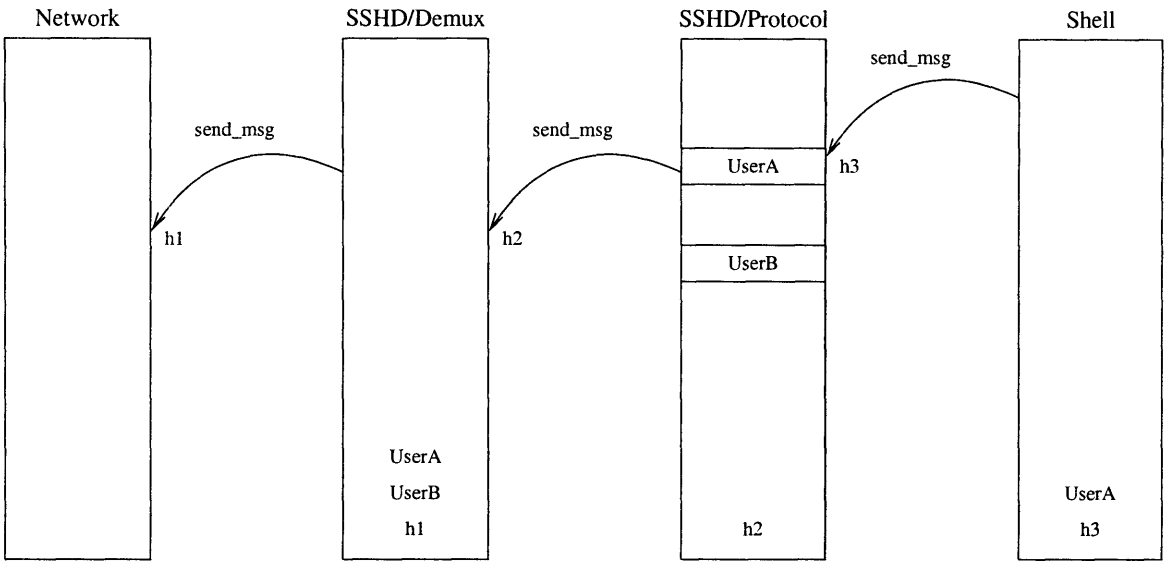

Figure 1-1: Different security domains in Asbestos. Each process has its own address space and default security domain. In this diagram, every process has the privilege of sending messages to the processes to the left of it. In addition, the SSHD/Demux process can declassify user-private data so that it can be sent over the network. The SSHD/Protocol process has multiple sub-domains that can only be entered in response to receiving messages from either the SSHD/Demux or the Shell process. These sub-domains are effectively isolated from each other.

combination of shared memory and message passing than to write applications that use only message passing. Sharing memory is often a natural way to enable communication between two parts of an application. In cases where a message passing interface is difficult to implement, shared memory gives application designers an alternative to simply combining two different modules into one, violating the principle of least privilege.

Another problem with Asbestos is complexity. There are many different abstractions in Asbestos that use security labels in different ways. Process labels and labels on pages of memory can both accomplish the same tasks, yet page labels are strictly stronger as they can control access to all of the memory in a process. It is also possible for a process to control what information it is willing to accept using either **vm_restore** (Section 2.6) or receive labels (Section 2.3.1, yet **vm_restore** is strictly stronger. Asbestos is unnecessarily complicated in these respects, and using simpler abstractions can help.

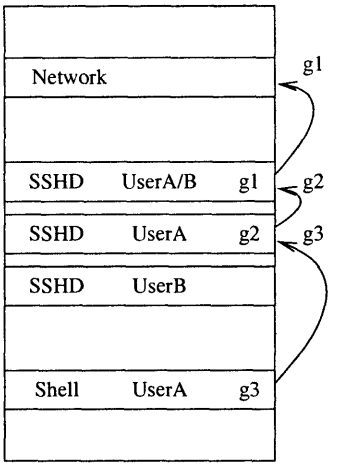

Figure 1-2: Security Domains in Acetone. There is only one address space, and different security domains correspond to regions of memory. Each domain has access to some data and the use of other gates. The arrows represent a thread using a call gate. The SSHD/Demux domain can declassify UserA-private data so that it can be sent over the network.

1.5 Approach

Acetone solves the problems presented above by using a unified address space, multiple threads of execution, and call gates that can transfer information into different security domains.

Acetone provides isolation mechanisms that make it easy to create as many domains as desired. It is possible to transfer data to another security domain by using a *gate*. These gates provide access to additional resources, but also ensure that these resources are only used in safe ways. It is also possible for domains to directly share memory, which can be an easier way to move data from one domain to another. Domains and gates are both lightweight primitives, allowing a single application to setup as many security domains as it needs with minimal resource usage.

Figure 1-2 shows how the same SSHD server could be implemented in Acetone. In Acetone, there is only one address space, and various addresses are in different domains. Each domain has access to some memory and some gates. It is also possible for each domain to be labeled with a specific user, meaning that the domain contains user-private data.

1.6 Challenges

Acetone presents an interface where every memory operation appears to be checked by label rules, but the cost of actually performing these checks on every memory access is prohibitive for modern hardware. A major challenge is to achieve high performance while ensuring that protection domains are never violated. Acetone avoids some of this cost by using hardware support for memory protection domains. However, a memory protection domain is a relatively large data structure that is expensive to create and update. Acetone must support many different protection domains, so it is important that these operations can be performed quickly. Acetone must support cheap creation of new domains that do not cause many faulting memory references.

In addition to the performance challenges, the API provided by Acetone must be simple and secure. It should be easy to define different security domains, and Acetone must enforce hard modularity between them. This means that a failure in one domain cannot directly cause failures in another, non-overlapping, security domain. It should be clear which domains can make accesses to a given domain.

To make the information flow guarantees in Acetone as strong as possible, there should be no direct communication channels that are not explicitly allowed by the security policy. This means that the availability of a specific resource cannot be used to communicate information.

This system should be as simple as possible. This includes making the trusted computing base small and also having a small number of system calls. Having these features implemented in an easy to use manner makes it much easier for an application writer to design their application using the principle of least privilege.

1.7 Limitations

Acetone is not yet fully implemented, and it has design difficulties that still need to be investigated. Only the core system calls, memory protection, and basic test applications have been implemented. Without more applications, and libraries to help

support those applications, it is difficult to evaluate all aspects of Acetone's design. However, Acetone demonstrates that Asbestos labels can be used with an acceptable performance overhead to provide protection between different components of an application.

Also, as discussed in Section 6.1, some basic desirable features such as accounting for and imposing limits on resource usage are not possible in the current design.

1.8 Contributions

This thesis has two main contributions. The first is a simplified operating system design, Acetone, that addresses weaknesses in Asbestos. The second is a high performance implementation of memory protection using Asbestos labels that allows new protection domains to be quickly created.

Chapter 2

Background

2.1 Abstractions

Operating system kernels make functionality available to user applications through various abstractions.

Abstractions in most UNIX operating systems include processes, user identifiers, file descriptors, path names, and virtual memory operations. By using these abstractions, applications can access various services provided by the kernel or other applications. Many of these abstractions are actually quite complex. A process is really the combination of a virtual address space, a current register state, possibly many kernel threads, a set of open file descriptors, and some security tokens such as user and group identifiers.

In a traditional microkernel architecture, abstractions generally only include processes, interprocess communication (IPC), and memory operations. Other services must be accessed through IPC.

Asbestos provides abstractions of handles, labels, memory, and processes. As in microkernel designs, these simple abstractions can be used to provide support for more complex features such as networking and filesystem access.

2.1.1 Handles

Asbestos labels serve two main purposes. They serve as communication endpoints. In addition, they can act as a capability or as a security label for mandatory or discretionary access control. Asbestos handles are simply 61-bit numbers that refer to a security label component and a communication endpoint. These numbers are ephemeral (they are not assigned the same values after reboots), unique (the same handle will not be reused), and unpredictable.

This code fragment demonstrates an application generating a new handle:

```
handle_t h;
r = sys_new_handle(&h, ...);
```

The **new_handle** system call creates a new handle and grants it to the current process. The kernel associates a security label with this new handle. It is also possible for the kernel to make this handle be a communication endpoint; this allows the process to receive messages that are sent to this handle.

At any time, any process can attempt to send a message to a handle. If the handle is set up as a communication endpoint, then the message is sent, provided that all checks pass. These access control mechanisms are discussed further in Section 2.2.

There are also system calls that allow some properties of a handle to be changed. The **handle_transfer** call can be used to change the communication endpoint of a handle.

All handles are reference counted by the kernel. When no processes have access to the handle, all kernel resources associated with the handle are freed. However, the actual number of the handle is never reused, as it is possible that applications still refer to that number even though they do not have any privileges with respect to that handle.

2.1.2 Messages

Messages are the basic packets of information transmitted in Asbestos. All communication between different processes must be done by sending messages. Messages

are an in-order sequence of bytes, and have an associated *destination handle*, *type*, *code*, *ID*, optional *verification label*, optional *reply handle*, and optional *payload*. Essentially the type, code, ID, and payload make up the data content of the message. The verification label can allow the sender to prove that it has certain security properties.

In general, a message's code is just used for very small payloads, where only a simple error code must be reported. The ID field is used to match up requests and replies. Asbestos defines common message types that are used to standardize communication between different modules and make virtualization easier.

Message types include:

`READ`, `WRITE` Requests data from a handle or sends data to a handle.

`LOOKUP` Requests a new handle for a related object. For example, for a directory handle, `LOOKUP` requests can be used to receive the handles for files within that directory.

`CONTROL` Makes a request that does not easily map into the other types.

`*_R` Replies to a request. Each message type has a reply type as well, used for responses. For instance, `LOOKUP_R` messages generally grant a handle to the process that issued the lookup request.

Messages are stored in a FIFO in the kernel until they are delivered. Each process has its own FIFO of messages waiting to be delivered. All access control checks on sending the message are done at send time. However, the receiving process's security label will not be updated until the message is actually received. Virtual memory tricks are used whenever possible for message delivery. For small messages, the actual message data is just copied onto the page that contains the message header. For larger messages, pages are mapped copy-on-write.

P, Q	Processes	$\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	Label levels, in increasing order
h, *dest*	Handles	L, C, D, V, E	Labels (functions from handles to levels)

$L_1 \leq L_2$	Label comparison: true if $\forall h, L_1(h) \leq L_2(h)$	P_S	Process P's send label
$\max(L_1, L_2)$	Maximum label: $\{h\, k \mid k = \max(L_1(h), L_2(h))\}$	P_R	Process P's receive label
$\min(L_1, L_2)$	Minimum label: $\{h\, k \mid k = \min(L_1(h), L_2(h))\}$	h_R	Handle h's receive label
$\text{owned}(L)$	Owned-handles label: $\{h \star \mid L(h) = \star\} \cup \{h\, \mathbf{3} \mid L(h) \neq \star\}$		

Figure 2-1: Asbestos notation.

2.2 Asbestos Labels

All access control and security in Asbestos is implemented using *labels*. The information stored in a label is flexible enough to allow a wide variety of access control policies to be implemented.

A label is a function that maps *handles* to *levels*. Possible levels include $\star$, **0**, **1**, **2**, and **3**. Labels are described as a list of (*handle*, *level*) pairs, and a default level which is assigned to all handles that are not in the list. For instance, the label $\{h_1\, \mathbf{0}, h_2\, \mathbf{1}, \mathbf{2}\}$ has a default level of **2**. The default level, **2** in this case, appears at the end of the list and applies to all handles not explicitly listed.

Figure 2-1 shows the basic notation and operations that can be done on labels. The basic operations include $\leq$, max, and min. As shown in the figure, these operations are run on each of the label components individually to produce the result.

2.3 Label Policies

Labels are used to restrict access to various resources. Processes, pages of memory, and handles all have rules associated with them that dictate how a given process can interact with them.

2.3.1 Processes

A process P has two labels associated with it: a *send label* (P_S) that stores the current process's capabilities and access restrictions, and a *receive label* (P_R) that restricts which access restrictions the process is willing to accept.

Process A is only allowed to send a message to process B if

$$A_S \leq B_R.$$

This check is only used to determine if a message can be delivered. If it can be delivered, then information flow restrictions are sent along with the message by updating

$$B_S \leftarrow \max(A_S, B_S).$$

Together, the receive restriction and the forwarding of taint information make up a basic information flow system, as has been shown in many previous designs [5, 13, 15].

These label rules imply that lower levels in the send label are more permissive. Handles that have low levels associated with them in the send label act as either capabilities or as low levels of tainting. This means that low levels in the send label are more permissive. Ultimately, if a process has every handle at 0 or $\star$, it can send a message to any other process.

In the receive label, lower label components are more restrictive. A label component at a low level in the receive label means that the process will not receive messages from a sender at a higher level. This has the effect of restricting the communication between the processes, and also of limiting what restrictions a process is willing to accept on its send label.

2.3.2 Effects of Labels

By setting send and receive labels in different ways, it is possible to enable many different policies. If all label components are either at the default level or the $\star$ level, then labels end up acting like capabilities; they can be freely granted to anything that the process can communicate with. If the other label levels are used, it is possible for the labels to enforce a variety of information flow constraints.

By lowering a process's receive label, only processes with explicitly lowered send

labels can send messages to the given process. If a process's send label is higher than the default receive level, the process will not be able to send messages to other processes unless they have explicitly raised their receive labels. As no user can modify the receive labels of the I/O devices on the system, this effectively stops a process from releasing any of its data to anything outside of the system.

2.3.3 Effective Labels

A process has some control over how its messages are sent. It is possible for a process to send a message that is less permissive than the process's current label. These *effective labels* allow a process to use a discretionary security policy by choosing what restrictions and privileges are associated with the message.

When sending a message, a process can provide a *verification label* V and a *contamination label* C_S. Using these, the kernel creates effective send and receive labels:

$$E_S = \max(P_S, C_S), \quad E_R = \min(Q_R, V).$$

These labels are then used instead of the process labels for computing the label check and the label contamination parts of the send process. This does not violate any security policies because these labels are all more restrictive than the process labels. The kernel gives the receiver a copy of V, allowing the sending process to prove that it has a label at a low level, without granting any privilege.

As an example, consider a trusted multi-user file server. This system has two handles for each user. One handle, u_I, is used for the privilege to access a user's resources. Another handle, u_C, contaminates all user private data. A process acting on behalf of a user would have a send label of $\{u_I\,\mathbf{0}\}$ and receive label set to $\{u_C\,\mathbf{3}\}$. The receive label is set to a high level so that the process can receive data that is private to user u. Once the process has received a message containing user u private data, its send label is $\{u_I\,\mathbf{0}, u_C\,\mathbf{3}\}$. This new send label prevents the process from communicating with non-u processes, as they will have receive labels with $\{u_C\,\mathbf{2}\}$. When the file server receives a message, it checks the verify label, V, only accepting

the message if $V(u_I) \leq 0$.

Asbestos provides an additional method of restricting which messages a process can receive by allowing additional label constraints to be placed on individual handles. These are called *handle labels* and can be set by the process that receives messages sent to that handle. The handle label restriction and the process label restriction are both checked before delivering a message to a handle. When a process P sends a message to process Q over handle *dest*, the effective receive label is:

$$E_R = \min(Q_R, dest_R, V).$$

2.3.4 Ownership and Decontamination

The $\star$ level allows processes to distribute handle access and to declassify information. A process P that has $P_S(h) = \star$ can be considered to *own* handle h. This process then has the ability to *decontaminate* data with respect to h. This decontamination can be done in two ways. It can be done when the process receives a message. No matter what the effective send label of the message is, P's send label will stay at $P_S(h) = \star$. Process P can also modify other process's send labels with respect to h, effectively declassifying an entire process. This is done by the sender of a message providing two *decontamination labels* with each message. One label, D_S, is used to lower a processes send label and the other, D_R, is used to raise its receive label. Both of these changes add privileges to the process that receives the message, and hence they require that the sending process have $P_S(h) = \star$. More formally, once the message is actually delivered, the kernel sets

$$Q_S \leftarrow \max(\min(Q_S, D_S), E_S) \text{ and}$$
$$Q_R \leftarrow \max(Q_R, D_R),$$

to actually decontaminate the process Q that receives the message.

Making a process's receive label more permissive can allow the process to become tainted, which restricts its ability to communicate with other processes. Therefore,

an additional check is done before changing a process's receive label. The kernel will not deliver a message unless $D_R \leq dest_R$. This check allows a process to control what changes are made to its receive label.

A process is automatically made the owner of any handle that it creates because **new_handle** sets $P_S(h) = \star$ when it creates a new handle. In addition, the kernel sets the new handle's receive label $h_R(h) = 0$ so that only processes that have been explicitly granted access to the new handle will be able to send messages to it. Every call to **new_handle** returns a new handle that has never been used before. This ensures that the process that creates a handle starts out as the sole owner of that handle. To change the receive label of a handle, a process can call **set_handle_label.** Also, a process can use the **setlabel** system call to reduce its own rights to a handle or to allow messages sent to that handle to be delivered to another process.

These ownership, creation, and declassification primitives allows access control and information flow systems to be built in a completely decentralized manner that does not require global trust.

2.4 Examples

This section presents a few examples of how Asbestos labels can be used to provide different access control policies. The fact that this primitive, the Asbestos label, can implement such a wide variety of policies shows how powerful it is.

2.4.1 Process Isolation

This example focuses on a process P that will create a process Q that is isolated from the rest of the system. This situation could arise when a web browser wishes to run a game that is downloaded off of the internet.

One way of achieving this goal is by restricting Q's send label so that it can only send messages to P. It will still be possible for Q to receive messages from any process in the system, but all outgoing messages must be sent to P. This is accomplished by having P create a new handle, j, and increasing Q's send label for

j to higher than the default receive label. Process P then raises its own receive label for j so that it can receive messages from Q. Once these steps have been taken, the processes are set up as follows:

Labels	P	Q	Others
Send	$j\star$	$j\,\mathbf{3}$	$j\,\mathbf{1}$
Receive	$j\,\mathbf{3}$	$j\,\mathbf{3}$	$j\,\mathbf{2}$

As a different policy, P may wish to completely isolate Q by only allowing it to send message to P and only receive messages from P. This restricts as the above policy does, but it also generates a new handle k that is used to restrict which processes Q can receive messages from. By setting $Q_{\mathrm{R}}(k) = Q_{\mathrm{S}}(k) = \mathbf{0}$, Q can only receive messages from P. The result of these operations is shown:

Labels	P	Q	Others
Send	$j\star, k\star$	$j\,\mathbf{3}, k\,\mathbf{0}$	$j\,\mathbf{1}, k\,\mathbf{1}$
Receive	$j\,\mathbf{3}, k\,\mathbf{2}$	$j\,\mathbf{3}, k\,\mathbf{0}$	$j\,\mathbf{2}, k\,\mathbf{2}$

This setup means that Q can only receive messages from senders that have a send label level for k at 0 or lower. P is the only process that has that property, so it is the only process that can send messages to Q.

2.4.2 Multi-level security

A multi-level security (MLS) system is one where each domain is restricted by a notion of current and maximum security level. In Asbestos, these domains just correspond to different processes. A "maximum level" corresponds to security clearance, and "current level" corresponds to the sensitivity level of information that the current process has actually been exposed to. For this description, consider the security levels of *secret* and *top secret*. The rule for message sending is that process P can send a message to process Q if P's current level is less than or equal to Q's maximum level. MLS systems also allow for some processes to be part of the TCB[1]. These processes have the ability to change their current security level at will.

[1]Trusted Computing Base

Many MLS systems are statically defined by the system that implements them. Each secrecy level is defined, and every application that uses these secrecy levels is restricted to using just the predefined set. Asbestos labels make it possible for applications to define their own system with an arbitrary number of different secrecy levels. The systems can also be virtualized such that different groups of processes can be participating in different MLS schemes simultaneously.

This system is implemented in Asbestos by having a trusted process M that is in charge of the MLS space. This process creates handles corresponding to each different security level, therefore it possesses $s\star$ and $t\star$ corresponding to the secret and top secret levels. Any other trusted services, such as a multi-level filesystem, are granted access to these handles at the $\star$ level as well.

When a process enters the MLS system, its current level is set to *unclassified*, and its maximum level is set according to what authentication that process can provide. Once a process's current security level is changed, it can no longer send messages to processes that have no access to that security level. If a process S receives some secret data in a message, that message will also have a contaminate argument that changes S's send label, restricting what processes it can send messages to. For instance, when the multi-level filesystem sends a message containing secret data to process S, it also uses the contaminate argument C, setting $S_S(s) = \mathbf{3}$, resulting in the following labels:

Labels	U	S	T
Send	$s\,\mathbf{1}$	$s\,\mathbf{3}$	$s\,\mathbf{3}$
Receive	$s\,\mathbf{2}$	$s\,\mathbf{3}$	$s\,\mathbf{3}$

If a second process T, with maximum security level "top secret," then receives a message from S, T will also end up with send label $\{s\,\mathbf{3}\}$, causing T to only be able to send messages to processes with maximum security level of secret or top secret. If top secret information is transmitted to T, then the processes will have the following labels:

Labels	U	S	T
Send	$s\,\mathbf{1}, t\,\mathbf{1}$	$s\,\mathbf{3}, t\,\mathbf{1}$	$s\,\mathbf{3}, t\,\mathbf{1}$
Receive	$s\,\mathbf{2}, t\,\mathbf{2}$	$s\,\mathbf{3}, t\,\mathbf{2}$	$s\,\mathbf{3}, t\,\mathbf{3}$

2.4.3 Discussion

These two examples show how powerful the Asbestos labeling mechanism is. The one label primitive, combined with two different process labels and the concept of ownership, can be used to implement a wide variety of access control policies in a decentralized manner.

2.5 Implementation

When labels are visible to user processes, they are simply treated as an array of handle-level pairs plus a default level. The kernel represents labels as array of pointers to internal structures that store information about each handle. Many tricks are used to minimize memory usage, including sharing, copy-on-write, and using low-order bits in pointers to store the level.

2.5.1 Processes

Processes in Asbestos are quite similar to processes in other operating systems. A process has an isolated address space. The bottom 3.75GB of the virtual address space can be used by user applications. The top 256MB is reserved for the kernel. There is no shared writable memory. This is important because all interprocess communication happens through message passing, and label rules can be applied. If shared memory was allowed, that might enable a new channel of communication.

In addition to the address space, a process has a thread of execution. This is basically a copy of all of the registers, and allows the kernel to switch between different processes. A process also has a page fault handler address and a saved address of an exception stack. These are just used for user-level page fault handling.

As discussed above in section 2.2, a process has a send label and a receive label, which are used for dictating whether or not the process can send and receive messages.

Finally, a process can have a saved copy of everything listed above. This allows

the process to receive a message that is tainted with sensitive information, handle it, and then restore to the untainted state. The end result is that there are essentially many different security domains running within one process.

2.6 vm_save/vm_restore

So far, labels, and therefore domains, have only been discussed at the process granularity. As demonstrated by the example of the SSHD server, there are times when one application wishes to have many different security domains. This section presents functionality that allows a process to have many *label spaces* defined within it. These label spaces can only be accessed when the process receives a message with the correct label for accessing a label space. Checkpoint and restore methods are used to prevent information flow between different label spaces.

2.6.1 Design

The design goal of these label spaces is to support multiple domains at a lower cost than one process per domain. The mechanism for this design is motivated by event-driven architectures that are used in many fast servers [12, 16, 21, 22]. These services use a simple scheduling loop that looks like:

```
while (1) {
    get_next_event();
    process_event();
}
```

The advantage of this loop is that no state is implicitly carried from one event to the next. This is ideal for Asbestos's design, as it should be impossible for one event to implicitly communicate information to the next event. Note that by default, the first tainted message that this server receives will taint the entire process, including all future messages that the server receives. If we modified this loop to instead fork once per iteration, and have the child handle the event, it would have the label properties

that we desire. However, it would not be possible for *any* state to be communicated from one event to another, even if they were both tainted.

In order to allow state to be maintained from one even to another, some heap pages can be marked as accessible only to a specific subprocess. These pages are not accessible to other subprocesses. Any other changes to memory caused by an event are all done copy-on-write in order to avoid communication of information to other subprocesses.

Finally, subprocesses are defined in terms of their *send label*. Each page of memory is marked with the label of the subprocess that can access it. As subprocesses are defined by their send labels, it is only possible to gain access to a one is by sending the process a message with the correct contaminate and declassify arguments.

This functionality is implemented with the **vm_save**, **vm_restore**, and **page_taint** system calls. The **vm_save** system call essentially checkpoints a process, saving its register state and address space and then waiting for a message. When a message is delivered, the page-table of the process is manipulated to mark pages copy-on-write or normal access according to the new send label and the memory page labels. Once the subprocess is done handling the event, it calls **vm_restore**, which jumps back to the process state as of the **vm_save** call. Finally, a subprocess is allocated by the untainted process by calling **page_taint**, which marks a page with a given label.

send($dest, C_S, D_S, V, D_R$, data)
Let Q be *dest*'s controlling process
Let $E_S = \max(P_S, C_S)$
Let $Q_{newR} = \max(Q_R, D_R)$
Let $E_R = \min(Q_{newR}, dest_R, V)$
Let $Q_{own} = \text{owned}(Q_S)$
Requirements:
(1) $E_S \leq E_R$
(2) $D_R \leq dest_R$
(3) If $D_S(h) < \mathbf{3}$, then $P_S(h) = \star$
(4) If $D_R(h) > \star$, then $P_S(h) = \star$
Effects:
Grant D_S, contaminate with E_S, then restore owned handles
$Q_S \leftarrow \max(\min(Q_S, D_S), E_S)$
$Q_S \leftarrow \min(Q_S, Q_{own})$
$Q_R \leftarrow Q_{newR}$

new_handle(L)
Let h be an unused handle
Effects:
$h_R \leftarrow L$
$h_R(h) \leftarrow \mathbf{0}$
$P_S(h) \leftarrow \star$
Return h

set_handle_label($dest, L$)
Requirement:
dest was created by P
Effect:
$h_R \leftarrow L$

Figure 2-2: Some Asbestos label operations. P is the calling process.

Chapter 3

Design

The core of Acetone's design is the system call interface that applications use to interact with the operating system. In addition, this chapter covers the design of the virtual memory system in Acetone.

3.1 High Level Overview

Acetone expands on the original Asbestos design by adding support for shared memory between different applications and simplifying the different security domains in the system. By reducing the number of abstractions provided, and the number of ways that they can interact, Acetone is a simpler system than Asbestos yet still achieves the primary goal of allowing application developers to easily define small domains.

3.1.1 Abstractions

Acetone's design is built on a few application-visible abstractions.

Security labels allow arbitrary security domains to be easily defined. By setting up the appropriate labels, applications can be set up to enforce a wide variety of security policies. Other abstractions in the system all have labels associated with them in order to restrict access.

Memory pages store application data. Pages are uniquely identified by the virtual address that they are mapped at. Each page has a label that controls what domains can read or write to that page.

Threads run programs. Each thread has a current security label which defines the security domain that the thread is running in. Each thread can only access resources that are in its domain.

Call gates allow threads to transfer from one domain to another. A thread can request access to additional resources by making a request through a call gate. Each call gate has one label that restricts which threads can access that gate, and another label that represents the additional privileges gained by using that gate. A gate automatically sets the program counter of the thread that uses the gate to the entry point of the gate itself.

In this way, a gate in Acetone acts much like a handle in Asbestos because both abstractions allow a procedure to access resources in a different domain, but in a restricted fashion. The access to the different domain is limited because the program counter is set to the address associated with the gate.

3.1.2 System Calls

Acetone provides system calls for each abstraction.

mem_alloc, mem_free, mem_taint - Manage memory. Described in section 3.4

new_gate - Creates a new call gate. Described in section 3.5

jump_gate, fork_gate - Allows a thread to move into a different security domain. Described in section 3.3

exit, self_taint - Destroy or change the current thread's label. Described in section 3.3.

3.2 Unified Address Space

Acetone provides only one address space. Because all threads run in a single address space they can easily share data. A virtual address is sufficient to uniquely identify a page of memory. It is unnecessary to know which process or address space that address is associated with. This one global name makes sharing easier because every entity in the system agrees on the name of a given resource.

A possible disadvantage of using a unified address space is that the kernel can no longer allow an application to specify an address where it wants to allocate memory. This is not allowable because allocating memory at a specific address effectively communicates information to every other domain, as future requests for that address will fail. Therefore, specifying that memory should be allocated at a specific address could only be done by a completely untainted application.

Acetone's solution to this problem is that the kernel allocates memory at a random address. As long as the virtual address space is large enough, this makes it difficult to communicate any information through the availability of a given address. However, this design means that all applications must support arbitrary load addresses. This is desirable on any system that uses a unified address space. This feature can be easily implemented by using PC-relative addressing or by using a dynamic binary loader that rewrites parts of the binary to reflect the address that it is loaded at.

3.3 Threads

Threads are somewhat like Asbestos processes, except more lightweight. All threads execute in the same unified address space, so it is not necessary to store a virtual address mapping for each thread. Each thread also has a security label. This label restricts what resources the thread can access, including memory and call gates, as described below in section 3.4.1 and section 3.5.

For thread management, the only primitive operations are jumping to call gates,

terminating the current thread, and a combination of forking the current thread and jumping to a call gate. By calling **jump_gate**, the current thread will enter the given call gate. The details of this are described in section 3.5. The current thread can be terminated at any time by calling **exit**. Another call, **fork_gate**, creates a new thread that is a copy of the current one. This new thread then immediately jumps to the given call gate.

It is also possible for a thread to reduce its privilege label by calling **self_taint**.

3.4 Memory Pages

Memory allocation has also been changed from Asbestos because a thread can no longer specify what address it wants memory to be allocated at. The only primitive memory allocation operations in Acetone are **mem_alloc** and **mem_free**. Allocations are of any size and the memory will be allocated in a continuous region of virtual address space. The address of the allocation is returned. Any thread that is allowed to write to memory is also allowed to free that memory.

In addition to system calls for allocating and freeing memory, Acetone provides a call that can change a label on a given page. A thread can call **mem_taint** only if it is currently able to write to the page, and will also be able to read from the page after the call is made. More precisely, if the current thread, T, with $T_{own} = \text{owned}(T_{label})$ (refer to Figure 2-1 for a review of label ownership), wishes to change a page's label, P_{label}, to P'_{label}, it must be the case that $\min(T_{own}, P_{label}) \leq T_{label}$ and $T_{label} \leq P'_{label}$.

3.4.1 Memory Protection

Each page of memory has a label associated with it. This label is similar to the label on a thread except that it cannot have any components at the $\star$ level. The $\star$ level is disallowed because it denotes ownership, and threads are the only primitives that can own label components. A thread, T, is allowed to read from a page of memory, P, if $T_{label} \leq P_{label}$. The thread T can write to the page if it can read from the page and $\min(T_{own}, P_{label}) \leq T_{label}$. The virtual memory system in Acetone provides an

interface that terminates any thread that makes a memory access that is not allowed by label rules.

3.5 Gates

In Acetone, the transfer of information from one domain to another is done through gates. A gate is identified by the address that the program counter is set to when the gate is called. Each gate has state associated with it that enables the called domain to safely receive information from the caller domain.

When a gate is called, Acetone grants the current thread new privileges, sets the thread's program counter to the gate's address, and changes the label on the memory containing the message so that it can be modified by the current security domain.

To create a new gate, an thread calls **new_gate** and gives it arguments corresponding to each of following properties:

`void *address` - The address of the gate's handler. This address is also the name of the gate itself. A thread must have write access to `address` in order to create a gate there. This constraint prevents information from being communicated from the availability of specific gates.

`label_t *minAllowed` - A thread and a message must have labels that are less restrictive than `minAllowed` in order to call this gate. More explicitly:
$\max(T_{label}, M_{label}) \leq H_{minAllowed}$
This property functions like the label on a page. A gate, G, is accessible to every thread, T, where $T_{label} \leq G_{minAllowed}$.

`label_t *declassify` - The label that is used to declassify the incoming thread and message. This also has the effect of preventing anyone but the receiver from modifying the message after it has been sent.

`bool_t newStack` - A flag that indicates that a new stack should be allocated when a thread uses this call gate. This exists solely as a performance optimization.

`bool_t singleUse` - A flag that indicates that this gate should only be used one time. When a single-use gate is used, the thread is tainted with `minAllowed` because otherwise the availability of this gate could communicate information between two disjoint domains. Tainting with `minAllowed` ensures that all threads that can call this gate have equivalent security domains, ensuring that it is acceptable for information to be shared between them.

3.5.1 Using Gates

To send a message from one domain to another, the current thread calls

jump_gate_simple(void *gate, void *msg, size_t length).

In this case, `gate` is the gate that will be called, `msg` is a pointer to the page or pages of memory that the message is on, and `length` is the length of the message, and must be evenly divisible by the page size.

If the current thread (T) calls **jump_gate_simple**(G, P, 1 page), where G is a gate and P is a page of memory containing a message, the system call executes the following steps.

1. Check if the current thread has write access to P. If it does not, then return an error.

2. Check if the gate is in the current thread's domain ($T_{label} \leq G_{minSend}$). If it is not, then return an error.

3. $T_{label} \leftarrow \min(T_{label}, G_{declassify})$.

4. $P_{label} \leftarrow \min(P_{label}, G_{declassify})$.

5. $T_{program\ counter} \leftarrow G$.

The other registers (including the ones that contain the message and length) are left alone, allowing some data to be passed through the call gate in the registers themselves.

3.5.2 Message Sending

A common use of call gates is for sending messages between domains. In this case, it is common for a thread in one domain to send a message to another domain, and for that domain to eventually send a reply message back. The more general **jump_gate** system call is set up to help in this common case. This call is much like **jump_gate_simple**, except that it optionally creates a reply gate, and calls **self_taint** in order to restrict what privileges are usable by the code associated with the gate.

The system call is:

jump_gate(void *gate, void *msg, size_t length, label_t taint, bool_t withReply)

This call performs the following steps.

1. If a reply gate is requested, Acetone creates a single use reply gate that has the same program counter and stack pointer as the thread that called **jump_gate**.

2. **self_taint**(`taint`)

3. **jump_gate_simple**(`gate, msg, length`)

3.6 Hardware Support for Memory Protection

As described above in section 3.4.1, Acetone must ensure that every memory reference obeys label rules. As actually checking the label rules for a given address is quite expensive, it is desirable to cache the results of these checks. If these cached results are stored in a standard x86 page directory, it is possible to use the CPU's default memory protection mechanisms to check memory accesses against the cache of currently accessible addresses.

To actually check the label access rules, the kernel must store information about memory. For each page of virtual memory, the kernel must be able to determine if that page is mapped, what physical page it maps to, and the label of that page. This state is stored in two different structures. The first is the ppage array, which has one entry for each physical page of memory. This structure keeps the label of each page,

and a list of free physical pages. The second structure is a mapping from virtual page to physical page. This is saved in an x86 page directory and page tables.

The sections below describe algorithms to implement memory protection using x86 hardware. On x86, the page table structures keep track of both address translation and memory protection. For Acetone, the address translation is the same for every thread; only the address protection must frequently change. Because of this, these sections refer to the x86 hardware as having a current *virtual protection space* rather than virtual address space. This protection space is defined by the permission bits on the page table entries.

3.6.1 Simple and Slow

A simple algorithm that likely results in bad, but not unreasonable, performance is one that simply caches the results of label checks and flushes the cache whenever these cached results might no longer be valid. When an instruction attempts to access memory that is not currently in these cached results, the processor generates a page fault exception. The kernel handles this exception by terminating the current thread or adding a page of memory to the current protection space.

Initially, the entire protection space is set up to have no permissions. When a page fault occurs, the kernel page fault handler checks the label of the current thread and the label of the page of memory, and adds permission bits corresponding to that page if the access is valid. Whenever the currently executing label becomes more restrictive, all permissions to the virtual protection space are removed, resetting the system to its initial state.

This algorithm results in many unnecessary page faults. For instance, any time the processor switches threads, it is likely that the new thread's label will not be strictly more permissive than the previous thread's label. This implies that after almost every thread switch, all page tables need to be cleared and there will be a sequence of page faults. However, this simple design is still a dramatic improvement over checking every memory operation.

3.6.2 Reducing Page Faults

The performance flaw with the simple design is that it must recompute the same label check for the same page if the scheduler switched threads momentarily. It is possible to improve upon the above algorithm by keeping multiple cached virtual protection spaces.

The basic idea behind this algorithm is that the kernel keeps a mapping from labels to cached virtual protection spaces. Each virtual protection space is guaranteed to have no more rights than are specified by the label that maps to it. When a thread is scheduled to run at a given label, the kernel sets the processor's protection space to the cached protection space associated with that label. If no cached protection space is found, then a new one is created.

When a page fault occurs, the label of the faulting address is compared against the current thread's label. If the access should be allowed, the current virtual protection space is updated.

Once there are multiple virtual protection spaces, it becomes necessary to keep track of which spaces allow access to a given page so that when the page is unmapped, it can safely be removed from all of the cached protection spaces.

For each page, a list of cached protection spaces containing it is maintained. When the page is freed, this list is traversed and the page is removed from every cached protection space. This maintains the invariant that every protection space only has access to pages that are allowed to be accessed by the label associated with that address space.

3.6.3 Decreasing Creation Costs

A weakness with this design is that it is expensive to create new domains. Every time that a new label is used, a completely new protection space must be created. Each protection space will take up at least a few pages of memory, and the time spent initializing them can be significant.

The creation cost of new domains can be decreased by allowing a second kind of

protection space. The new kind of protection space, called a *sub-protection space*, is a structure that contains a pointer to another protection space, called the *parent* protection space, and a list of additional memory privileges. A sub-protection space's label is always less than its parent's label.

As in the simple algorithm above, when a page fault occurs, the kernel checks to see if the current thread has the necessary rights to access that memory. If it does, then the kernel updates the current protection space to include the new permissions. Once a sub-protection space's list of changes grows past a certain size, the structure is replaced by a full protection space.

When the kernel switches to a sub-protection space, it switches to the parent protection space and then applies the list of additional permissions to it. When switching away from this protection space, it removes the additional permissions. When a page fault occurs, the kernel first checks if the label rules allow the kernel to add the mapping to the parent protection space. If it cannot be added to the parent space, the kernel checks the label rules for the current protection space.

3.6.4 Finding Parent Protection Spaces

When creating a new protection space, it is desirable to find a suitable parent protection space. A protection space P can be a parent for a new protection space N if $N_{label} \leq P_{label}$. If no suitable space can be found, the empty protection space can be used, however this will likely result in many page faults. Acetone keeps one cached protection space for each gate and each currently running thread. When a thread changes its label or uses a call gate, these cached protection spaces are checked to see if they can be used as a parent for the new protection space. It is possible that using a more sophisticated cache could be beneficial; this is discussed in Section 6.1.2.

Chapter 4

Evaluation

This chapter evaluates the Acetone design both as a stand-alone design and in comparison to the original Asbestos design. First, it makes an argument about the simplicity, and therefore security, of Acetone as compared to Asbestos. Second, a covert channel analysis of Acetone is presented. Finally, actual performance results from different memory protection schemes are presented.

4.1 Simplicity

Acetone attempts to use the fewest and least complex abstractions possible in order to allow application developers to easily use the principle of least privilege.

In Asbestos, the procedure to set up, use, and deallocate a subprocess is as follows:

1. First, the server must be running in a **vm_save/vm_restore** loop listening for messages.

2. To start a new connection, the server must receive an untainted message that informs the server that a tainted connection will be arriving soon.

3. In response to this message, the server allocates a virtual address space range with **page_taint** for the tainted connection. Allocating this range sets up a subprocess that the server cannot directly access.

4. The connection is now used by sending messages with the appropriate labels to the server, and the server can respond to each message and call **vm_restore** in order to prepare to receive other messages.

5. Once the connection has finished, another *untainted* message must be sent to the server saying that the connection has closed. The timing of this message is a communication channel between the tainted connection and the untainted subprocess in the server, so it is generally sent a random amount of time after the connection closes in order to minimize the amount of information leaked.

6. Once this message has been received, the server frees the virtual address space range with **page_taint**.

This entire procedure places many requirements on the client that interacts with the server, involving both tainted and untainted messages.

In Acetone, the above situation is simplified because no untainted messages are required, and it is not necessary to preallocate resources for a new subprocess. This simplification is possible because memory allocations happen at random addresses and therefore do not communicate information. In Acetone, steps 2, 3, and 5 are not required. In addition, step 6 is a bit more natural in Acetone's design because a special system call is no longer necessary. Freeing of a tainted connection's data can be done with the standard **mem_free**.

Another simplification is that in Acetone it is not necessary to have a separate process send messages with appropriate labels in order to communicate with subprocesses. As gates can have privileges associated with them, all of these features can be implemented with gates and threads, so all parts of a server can be implemented in one binary.

Another advantage provided by the Acetone messaging system is that there is no kernel buffering of messages. The only memory resources managed in the kernel are execution contexts, gates, security labels, and pages of memory. In Asbestos, the kernel also has to manage arbitrarily sized messages for an arbitrary amount of time. This is difficult for accounting because the memory is not technically under control

of either process. By removing this case, it should be easier to implement a resource accounting system, as discussed in Section 6.1.1.

The last argument for Acetone's improved simplicity is that it allows for shared memory. This does not actually make the operating system kernel any more simple, but it does allow some application code to be simplified because it is not necessary to transmit all information in carefully constructed messages. This argument is not conclusive though, as shared memory can also be bad for security. It is more likely that malicious code can cause damage to another module if the modules share a memory region than it is if the modules can only send messages to each other.

4.2 Covert Channels

All communication channels that are not explicitly allowed by label rules are based on resource starvation. It is theoretically possible for two threads to communicate by controlling the availability of physical memory, virtual address space, CPU time, network access, or any other resource that both threads have access to. Many of these channels could become much less threatening if resource accounting was used, as discussed in Section 6.1.1.

Asbestos provides direct timing channels because a process can only handle one message at a time. Essentially, information can be communicated by the precise time that a process calls **vm_restore**. In Acetone, this is impossible because a different thread handles each request concurrently. This could make synchronization for the server difficult, and in some cases it may be necessary to reintroduce that timing channel by adding a trusted lock server that must be granted access to each label component. However, in the common case, where each request is executed in a thread that runs in an independent security domain, the requests can actually be executed in parallel, and this timing channel is not necessary.

4.3 Performance

Acetone is implemented for common x86 hardware. First, this section discusses some difficulties in using x86 paging mechanisms for applying Acetone's memory protection. Then performance results from Acetone running on real hardware are presented and discussed.

4.3.1 Hardware Paging

It is clear that the x86 page protection and translation mechanisms are not ideal for Acetone. On x86 platforms, both page translation and protection are done with the same mechanism. Ideally, there would be two separate mechanisms for these two features. Acetone would fully take advantage of this because it would have just one address translation structure that could be used by all execution contexts. There would only be separate address protection structures for each individual execution context.

In addition, the hierarchical page table mechanism in the x86 architecture is not ideal for a sparsely filled address space. As allocations tend not be close together in the virtual address space, each memory allocation often requires a new page table to be allocated and used. This effect can be mitigated by allocating larger regions of memory at one time, or by reducing the randomness of memory allocations.

4.3.2 Method of Testing

In order to evaluate the different memory system implementations, a few different microbenchmarks are run. These microbenchmarks analyze the cost of both creation of new security domains and the cost of switching between them. The hypothesis is that by caching protection domains and using sub-protection spaces discussed in Section 3.6.3, it is possible to have minimal overhead from creating, maintaining, and enforcing protection domains.

The first microbenchmark just sends a message (a page of memory) back-and-forth between two non-overlapping security domains. There is only one thread in

this test, and it simply calls a gate that is in the other domain, and creates a reply gate to be used to return. An analogous test on a UNIX system is to use pipes to send a byte of information back-and-forth between two processes.

The second microbenchmark measures the cost of creating a new domain, entering it, and returning from it. This test is analogous to a server running code on behalf of a user that it has never seen before. It creates a new label component, calls a gate in a different domain, providing the new label component as a `taint` argument. This new domain has the privileges granted by the call gate, but still has the restriction imposed by this taint argument. The new domain reads authorized data, writes data to a reply message, and then returns through the reply gate. The UNIX version of this test calls **fork** on every iteration and the child receives a request from and responds to the parent.

The tests were each run in three different configurations. The first was on a version of Acetone with the very simple memory protection scheme discussed in Section 3.6.1. The second run was on Acetone with the optimized memory protection scheme discussed in Section 3.6.3. The last test was run on a machine running Linux 2.6.9. The tests of Acetone were run on an AMD 1500+ with 64MB RAM. The Linux tests were run on a Pentium M 1.3GHz with 512MB RAM. None of the tests are memory constrained, and the two different processors have similar performance characteristics.

4.3.3 Results

The results from these tests are shown in Figure 4-1. For the ping-pong test, each domain accesses the stack, the executable code, and the message. There are never any page faults caused by the message, because the **send** procedure ensures that it is already in the cached protection space. The simple protection mechanism therefore takes two page faults each time that a gate is used, one for the stack, and one for the executable code. This gives four page faults per iteration because each iteration uses an entry call gate and a return call gate. The simple protection implementation is so slow overall because of the large amount of time spent clearing out protection do-

Test	Protection Mechanism	Time	Page Faults
ping-pong	simple	31.9 μs	4
ping-pong	optimized	5.5 μs	0
new-domain	simple	30.1 μs	3
new-domain	optimized	6.3 μs	1
ping-pong	linux	4.0 μs	0
new-domain	linux	100.0 μs	1

Figure 4-1: Microbenchmark results running with both the simple memory protection mechanism and the more optimized implementation on an AMD 1500+ processor with 64MB RAM. Similar tests were also run on a 1.3GHz Pentium-M machine running Linux 2.6.9. The ping-pong test sent a message to an existing domain and waited for that domain to reply. The new-domain test is similar, except that each message is sent to a completely new domain. Each test was run 100 times and the results were averaged.

mains. The optimized protection implementation takes no page faults in steady state because it caches protection domains for both reply gates. Its overall performance is much higher than the simple protection scheme in the ping-pong test because it does not need to create new protection domain structures and does not take any page faults. The optimized scheme has very similar performance to Linux.

For the new-domain test, the simple protection scheme takes one fewer page fault per iteration because a new stack is being mapped for the new domain each time as part of the **send** call. The optimized protection scheme takes one page fault per iteration when accessing the executable code in the new domain. This fault occurs because this new sub-protection domain's parent is the null domain, and the sub-protection domain contains only the message and the newly allocated stack. The performance is still quite high for the optimized implementation because creating a new domain is easy: it is only necessary to allocate a small structure that stores the difference between an empty domain and the new domain, rather than storing a complete page directory. For Linux, the performance on this test is much slower as calling **fork** is an expensive operation.

Chapter 5

Related Work

Acetone is based on the Asbestos [14] operating system. Asbestos labels, applying labels to pages of memory, and the implementation of capabilities as label components are all ideas that Acetone took from Asbestos. In addition, most of the related work in [11] is relevant to this thesis as well.

Opal [3] is a single address space operating system similar to Acetone. The principle difference is that Opal defines a domain in terms of a list of capabilities, whereas Acetone uses Asbestos labels to define domains. Looking into optimizations and application design used in Opal could be valuable for Acetone's future progress.

Inferno [4] is an operating system that bases protection off of language level features instead of hardware protection. Like Acetone, all code is executed in a single address space. Inferno has the limitation that all code must be written in a specific type safe language. Singularity [7] is a recent operating system that also runs in a single address space and uses type safe languages to enforce protection.

The work in Paradigm Regained [20] shows that standard access control policies can be implemented on top of a capabilities based system. Asbestos labels, and therefore Acetone, implement some of these ideas.

Asbestos is similar to KeyKOS [6] and EROS [19]. The main difference is the use of Asbestos labels instead of basic capabilities in order to implement security policies. In addition, KeyKOS and EROS both put considerable effort into being persistent systems where everything can be checkpointed to disk. Asbestos and Acetone make

no effort to do this.

Gates in Acetone can be thought of as capabilities. Some believe that a capability system cannot implement mandatory access control policies [2]. This is not necessarily true, as long as other policies can be used to control the transmission of capabilities. Systems such as KeyKOS [10] and EROS [18] achieve mandatory access control policies by isolating processes into compartments and ensuring that any cross-compartment capability obeyed the mandatory access control guidelines.

Many systems have combined capabilities with extra checks on the use of capabilities. Systems have been built that use interposition [8], labels [9], and authority checks [1]. Acetone combines capabilities with decentralized labels.

Chapter 6

Future Work and Summary

6.1 Future Work

6.1.1 Accounting

One difficult challenge in operating system design is accounting for resource usage. Ideally, it should be possible to implement a resource allocation policy that can limit resource usage on a per application or per user level. Generally, resources that should be accountable include I/O resources such as network or disk, memory, and CPU time. The current Acetone design does not have any support for any sort of accounting. This is an area for future work.

The original Asbestos design also had little support for accounting. However, it did have distinct processes, which made it possible to kill certain applications after they had been started. It seems likely that the same features could be made available in Acetone if applications were given a certain label when they were started. This label would need to be shown to the system in order to allocate any resources. It would then be possible for a privileged program such as *kill* to find and free all resources that were allocated using that label. This approach is preliminary though, and requires a much more complete design.

6.1.2 Memory Protection Space Selection

The algorithm discussed in Section 3.6.3 does not ideally select cached protection spaces. In the case of specific tainting labels, for instance a label that taints data as being private to a specific user, this algorithm will not find a cached protection space, and will have to default to creating a new, fully restricted one. There are possibilities to improve both the selection process and the cache usage policies.

6.2 Summary

Acetone provides a simple interface for using Asbestos labels to implement security protection. Using very few abstractions, labels, threads, memory pages, and call gates, Acetone provides an interface that allows secure applications to be designed and implemented. The performance results from microbenchmarks show that Acetone can perform well and is a promising architecture.

Bibliography

[1] Viktors Berstis. Security and protection of data in the IBM system/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–252, May 1980.

[2] William Earl Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 291–293, September 1984.

[3] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. *SIGOPS Oper. Syst. Rev.*, 26(2):9, 1992.

[4] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *Proceedings of the IEEE Compcon 97 Conference*, pages 241–244, San Jose, 1997.

[5] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Oakland, CA, May 2000.

[6] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[7] Galen Hunt, James Larus, David Tarditi, and Ted Wobber. Broad new os research: Challenges and opportunities. In *To appear in Proceedings of the 10th*

Workshop on Hot Topics in Operation Systems, Santa Fe, NM, June 2005. USENIX.

[8] Paul A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 32–37, Oakland, CA, April 1987.

[9] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, April–May 1984.

[10] Key Logic. *The KeyKOS/KeySAFE System Design*, sec009-01 edition, March 1989. `http://www.agorics.com/Library/KeyKos/keysafe/Keysafe.html`.

[11] Max Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, June 2005.

[12] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June 2004. USENIX.

[13] Carl E. Landwehr. Formal models for computer security. *Computing Survels*, 13(3):247–278, September 1981.

[14] David Mazières, Frans Kaashoek, Eddie Kohler, and Robert Morris. Securing untrusted software with Asbestos. April 2004.

[15] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, 1992.

[16] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX*, pages 199–212. USENIX, June 1999.

[17] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[18] Jonathan S. Shapiro, Jonathan Smith, and David J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[19] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[20] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 558–563, Washington, DC, 1986. IEEE Computer Society.

[21] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, October 2003. ACM.

[22] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, October 2001. ACM.