SECURING UNTRUSTWORTHY SOFTWARE USING INFORMATION FLOW CONTROL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nickolai Zeldovich

October 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David Mazières) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Monica S. Lam)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dawson Engler)

Approved for the University Committee on Graduate Studies.

Abstract

This dissertation shows that trustworthy applications can be built out of largely untrustworthy code, by using information flow control to reason about the effects of code execution. Using this technique we construct a scalable distributed web server, in which most application code is untrusted and there are no fully-trusted machines or components.

Building secure applications from untrusted code requires safely executing arbitrary code on sensitive data, something that no current operating system provides satisfactory mechanisms for. To address this, we built a new operating system called HiStar that allows any user or application to specify precise data security policies. The HiStar kernel has a simple, narrow system call interface that enforces these policies by controlling information flow. HiStar provides a Unix-like environment with acceptable performance that is implemented in an untrusted user-level library but uses the kernel to enforce security, and runs a wide variety of Unix applications. The system has no notion of superuser and no fully trusted code other than the kernel. HiStar's features permit several novel applications, including an entirely untrusted login process, separation of data between virtual private networks, and privacy-preserving, untrusted virus scanners.

In a distributed setting, controlling information flow between processes on mutually distrustful machines poses another technical challenge. We addressed this by developing DStar, a framework for controlling information flow in distributed systems. DStar describes information flow restrictions associated with network messages, and allows multiple machines to enforce an overall information flow policy. DStar separates policy from trust by using self-certifying information flow restrictions, which include a public key in their name. HiStar applications can use DStar to safely run untrusted code across multiple HiStar machines. For example, a highly privilege-separated HiStar web server can take advantage of multiple HiStar machines for performance scalability by only adding a small amount of trusted DStar code. Even a fully-compromised machine can only subvert the security of users that use or have recently used that

machine. Finally, DStar eases incremental deployment, by allowing legacy systems to securely execute just the least-trusted code on HiStar.

Acknowledgments

I would like to thank David Mazières for his help and guidance in this work, and for being a great adviser. I am grateful to Monica Lam, Robert Morris, and Frans Kaashoek for teaching me about research. I would also like to thank Silas Boyd-Wickizer for his contributions to this work. Thanks to the Asbestos group at MIT and UCLA for providing inspiration and advice for the development of HiStar. I am thankful to Monica, Dawson Engler, Mendel Rosenblum, and Kunle Olukotun for taking the time to be on my committee. Thanks also to Constantine Sapuntzakis, Ramesh Chandra, and the members of the SCS group for many insightful discussions. Finally, I would like to thank my parents, my wife, and her parents, for their support.

Portions of this dissertation have been previously published in [59].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many serious security breaches stem from vulnerabilities in application software. Despite an extensive body of research in preventing, detecting, and mitigating the effects of software bugs, numerous errors remain. Experience has also shown that only a handful of programmers have the right mindset to write secure code, and few applications have the luxury of being written by such programmers. To make matters worse, the security of most systems ultimately depends on a large fraction of the code behaving correctly. As a result, we see a steady stream of high-profile security incidents [42].

How can we build secure systems when we cannot trust programmers to write secure code? One hope is to enforce the security policy of an application separately from the bulk of the application code. If security depends only on a small amount of code, this code can be verified or implemented by trustworthy parties, regardless of the complexity of the application as a whole.

This dissertation's thesis is to specify and enforce security policy in terms of how information can flow in a system, or in other words, in terms of how data in the system can be altered or disclosed. Traditionally, security policies have been specified in terms of what operations a program can invoke [20, 33], but reasoning about the effects of each operation quickly becomes difficult [15]. On the other hand, specifying security policy in terms of information flow allows reasoning about which components of a system may affect which others and how, without having to understand those components themselves. As a result, small amounts of trusted code can determine and control the security implications of executing much larger amounts of untrustworthy code.

Many typical security problems can be easily couched in terms of information flow. For example, protecting users' private profile information on a web site often comes down to ensuring that one person's

information (social security number, credit card, etc.) cannot be sent to another user's browser. Similarly, protecting against trojan horses means ensuring that network payloads do not affect the contents of system files. Likewise, protecting passwords means ensuring that whatever code verifies them can reveal only the single bit signifying whether or not authentication succeeded.

Specifying security policy in terms of information flow also makes it possible to objectively evaluate any security mechanisms. The security of any operation can be verified by first determining where information can flow as a result of the operation, and then verifying that such flows are consistent with the information flow policy. In contrast, security mechanisms in Linux or Windows are hard to evaluate objectively because they often change in response to application demands and lack a single underlying objective principle.

There has been a lot of prior work on information flow control, and covert communication channels have often been cited as a significant limitation [23]. We note that information flow control need not be perfect to be a valuable tool. Even with a relatively high-bandwidth covert channel of 1,000 bits/second, an attacker would take almost a full day to leak a database of 100,000 user records of 100 bytes each, a far cry from today's compromises of millions of records [42, 53]. Exploiting such channels usually requires fully compromising code that has access to all the data, which can be hard, and consuming large amounts of resources, which can be detected.

1.1 Contributions

This dissertation shows that it is practical to build secure applications from mostly untrusted code by using information flow control to enforce data security. For example, we build a scalable, distributed web server in which most of the code handling users' private information and passwords is untrusted, and only small amounts of partially-trusted code enforce the security of user data. The web server provides a familiar Unix environment for application code, and requires few to no changes to existing applications. Moreover, this web server has no fully-trusted machines or components—not even the kernel on any machine—which further minimizes the effects of any compromise.

Realizing our goal of building secure applications from mostly untrusted components required solving a number of technical challenges. No existing operating system provided satisfactory mechanisms for safely executing untrusted code with access to private user data. We addressed this shortcoming by developing a new operating system, called HiStar, which provides strong information flow control enforced by a kernel of under 20,000 lines of code. The kernel has a narrow system call interface that controls

information flow, and all access control decisions in HiStar are based purely on information flow. Every application in HiStar can define its own information flow control policies which will then be enforced by the kernel. HiStar has no notion of superuser, and no fully-trusted code other than the kernel, allowing application developers to decide what code they want to trust. Despite the lack of absolute superuser privileges, an administrator in HiStar can still manage the system's resources using a new container abstraction that separates resource allocation and revocation from all other forms of access control.

HiStar is able to make strong security guarantees by providing a simple, narrow kernel interface consisting of six object types. All objects in the system have a *tracking label* that defines how the information in each object can be observed or modified. Through its system call interface, the kernel provides only a small number of object operations with simple, well-defined semantics, making it possible to precisely track information flow. The kernel further takes into account all other information that goes into each operation, including privileges and resources, in order to avoid implicit information flows that lead to covert channels.

HiStar provides a shared-memory interface, in which the fundamental operations are reads and writes, as opposed to a more typical message-passing interface provided by many microkernels. Message-passing systems usually require bi-directional communication with trusted user-level services, such as a file server, in order to perform both read and write operations, making it difficult to enforce one-way information flow control in the kernel. Furthermore, message-passing systems often require implicit resource allocation for message queues, which can leak information. On the other hand, by providing shared memory access, HiStar can enforce one-way information flow control purely in the kernel, reducing the need for trusted user-level servers.

On top of HiStar's six kernel object types, we implement a familiar, Unix-like development environment in an untrusted user-space library, which runs standard applications such as gcc, gdb, OpenSSH, perl, and so on. Because HiStar controls information flow at the level of kernel objects, we can enforce strong information flow control for our Unix-like environment without having to understand the information flow semantics of complex Unix operations. The HiStar Unix library implements the superuser privileges of the root user purely by convention. Applications handling highly-sensitive data can choose not to grant their privileges to root, thereby reducing the amount of code with access to their data. At the same time, root can still manage the system by revoking the resources of recalcitrant users using the container abstraction.

Scalable applications require distributed systems of multiple machines for performance and reliability. However, an operating system such as HiStar can only enforce meaningful security guarantees on a

single machine. To build secure scalable applications using information flow control, we designed and implemented a distributed information flow control framework called DStar. The DStar design is completely decentralized, with no central authority or fully-trusted machines, which allows multiple mutually-distrustful entities to specify security policies for their data in a distributed system. Furthermore, a decentralized design avoids inherent performance bottlenecks and single points of compromise.

One problem with a decentralized design is that it can be difficult to determine when information is allowed to flow between two machines, and having the sender ask any other machine for permission may in itself leak information. DStar solves this problem by deciding whether information can be sent to another machine without any external information or communication, using only the delegation certificates to determine what machines are authorized to handle what data. To avoid any central authority, DStar embeds a public key in the name of each *category* that places information flow restrictions on data in the first place, and uses this public key to verify the signatures on delegation certificates.

Although our prototype implementations of HiStar and DStar likely contain covert channels, our security goal is to avoid covert communication channels inherent in the interface specification. Then, even if our implementations contain covert channels, we can incrementally mitigate the covert channels as necessary, without modifying the application interface, or affecting well-behaved applications. To avoid covert channels in specifications, we make explicit all information that goes into each operation, including trust relations, privileges, and resources.

In addition to building a highly-secure web server, we show how information flow control can be used to solve a number of other security problems as well. For example, a 110-line trusted wrapper program can ensure that a complex and untrusted virus scanner cannot leak any private files on HiStar. A novel login process on HiStar allows users to provide their own authentication code while ensuring that only one bit of information—whether authentication succeeded or not—is exposed, even if the user provides his password to malicious code. On a machine connected to both the Internet and an internal VPN, information flow control can ensure that data does not accidentally leak between the two networks.

1.2 Organization

The next chapter will discuss and evaluate the HiStar operating system in more detail. The DStar framework for distributed systems is described and evaluated in Chapter 3. Limitations of our approach and typical usage patterns are discussed in Chapter 4. Chapter 5 covers related work, and Chapter 6 concludes.

Chapter 2

Information Flow Control in an Operating System

The goal of the HiStar operating system is to enforce a data security policy when executing untrusted code with access to sensitive data. In particular, even if an untrusted application may be able to read some data, it should not be able to surreptitiously export this data from the system.

A good example of such an application is a virus scanner, which requires access to all of the user's private files, but should never disclose their contents to anyone else. Consider the recently discovered critical vulnerability in Norton Antivirus that put millions of systems at risk of remote compromise [36]. Suppose we wanted to avoid a similar disaster with the simpler, open-source ClamAV virus scanner. ClamAV is over 40,000 lines of code—large enough that hand-auditing the system to eliminate vulnerabilities would at the very least be an expensive and lengthy process. Yet a virus scanner must periodically be updated on short notice to counter new threats, in which case users would face the unfortunate choice of running either an outdated virus scanner or an unaudited one. A better solution would be for the operating system, HiStar, to enforce security without trusting ClamAV, thereby minimizing potential damage from ClamAV's vulnerabilities.

Figure 2.1 illustrates ClamAV's components. How can we protect a system should these components be compromised? Among other things, we must ensure a compromised ClamAV cannot purloin private data from the files it scans. In doing so, we must also avoid imposing restrictions that might interfere with ClamAV's proper operation—for example, the scanner needs to spawn a wide variety of external helper

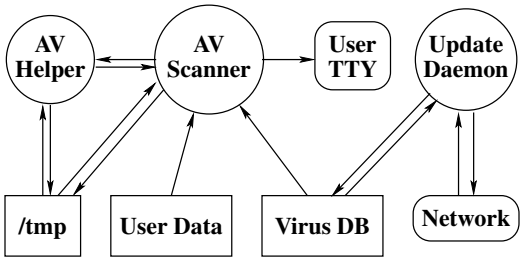

Figure 2.1: The ClamAV virus scanner. Circles represent processes, rectangles represent files and directories, and rounded rectangles represent devices. Arrows represent the expected data flow for a well-behaved virus scanner.

programs to decode input files. To understand the scope of the problem we are trying to solve, consider these few ways in which, on Linux, a maliciously-controlled scanner and update daemon can collude to copy private data to an attacker's machine:

- The scanner can send the data directly to the destination host over a TCP connection.
- The scanner can arrange for an external program such as *sendmail* or *httpd* to transmit the data.
- The scanner can take over an existing process with the *ptrace* system call or `/proc` file system, then transmit the data through that process.
- The scanner can write the data to a file in `/tmp`. The update daemon can then read the file and leak the data by encoding it in the contents, ordering, or timing of subsequent outbound update queries.
- The scanner can use any number of less efficient and subtler techniques to impart the data to the update daemon—e.g., using system V shared memory or semaphores, calling *lockf* on various ranges of the database, binding particular TCP or UDP port numbers, modulating memory or disk usage in a detectable way, calling *setproctitle* to change the output of the *ps* command, or co-opting some unsuspecting third process such as *portmap* whose legitimate function can relay information to the update daemon.

Some of these attacks can be mitigated by running the scanner with its own user ID in a *chroot* jail. However, doing so requires highly-privileged, application-specific code to set up the *chroot* environment, and risks breaking the scanner or one of its helper programs due to missing dependencies. Other attacks, such as those involving sockets or System V IPC, can only be prevented by modifying the kernel to restrict certain system calls. Unfortunately, devising an appropriate policy in terms of system call arguments is

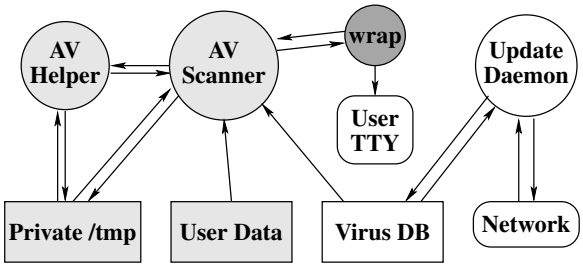

Figure 2.2: ClamAV running in HiStar. Lightly-shaded components are *confidential*, which prevents them from conveying any information to non-confidential (unshaded) components. The `wrap` process is strongly shaded, indicating that it has special privileges which allow it to relay the scanner's confidential output to the terminal.

an error-prone task, which, if incorrectly done, risks leaking private data or interfering with operation of a legitimate scanner.

A better way to specify the desired policy is in terms of where information should flow—namely, along the arrows in the figure. While Linux cannot enforce such a policy, HiStar can. Figure 2.2 shows our port of ClamAV to HiStar. There are two differences from Linux. First, we have labeled files with private user data as *confidential*. Labeling a file as confidential restricts the flow of its contents to other components that are not marked as confidential, including the network.

The second difference from Linux is that we have launched the scanner from a new, 110-line program called *wrap*, which has special privileges that allow it to export confidential data from the system. *wrap* can take the virus scanner's result and report it back to the user. The scanner process cannot read confidential user files without first marking itself as confidential. Once the scanner process becomes confidential, it can no longer convey information to the network or to the update daemon. So long as *wrap* is correctly implemented, then, ClamAV cannot leak the contents of the files it scans.

The information flow principles behind this type of isolation are not new. Several other systems have mechanisms capable of isolating an untrusted virus scanner, including SELinux [26], EROS [47], and in particular Asbestos [10], which inspired this work. HiStar's labels, which originated in Asbestos, have features resembling the language-based labels in Jif and Jflow [35].

Unlike these systems, though, HiStar shows how to construct conventional operating system abstractions, such as processes, from much lower-level kernel building blocks in which all information flow is explicit. HiStar demonstrates that an operating system can dynamically track information flow through labeling without the labeling mechanism itself leaking information, thereby avoiding a number of covert channels inherent in the Asbestos design. By separating resource revocation from access, HiStar also

shows how to eliminate the notion of superuser from an operating system without inhibiting system administration. A HiStar administrator can manage the machine's resources with no special rights to read or write arbitrary user data, whereas Asbestos has no mechanism for an administrator to reclaim resources. By providing a shared-memory kernel interface, unlike message-passing systems such as Asbestos, HiStar can enforce one-way information flow control purely in the kernel, without the need for trusted user-level servers. Finally, HiStar's Unix environment can run many existing applications, such as gcc, gdb, perl, and OpenSSH, with almost no source code modifications, which makes it easy for existing web applications to use HiStar.

The next section describes Asbestos labels [10], which HiStar uses to track and control information flow. Section 2.2 describes the design of the HiStar kernel interface, and how it avoids covert channels in its specification. Section 2.3 talks about some important implementation details, and Section 2.4 discusses how user-level applications, including the Unix library, make use of HiStar. A number of applications that run on HiStar are presented in Section 2.5, and they are evaluated in Section 2.6.

2.1 Tracking Information Flow

HiStar layers all operating system abstractions on top of six low-level kernel object types that will be described in the next section—threads, address spaces, segments, gates, containers, and devices. Every object has an Asbestos label that describes how that object's information can be observed or modified. For active "agent" objects, such as threads, which can perform actions, the label also specifies the privileges held by that agent.

The kernel provides a small number of well-defined operations that act on kernel objects. Every time a thread makes a system call or triggers a page fault, information can potentially flow between the current thread and other objects in the system. In all such cases, the kernel makes sure that this information flow is permitted by the labels of the current thread and the other objects. If the labels prohibit the information flow that would be caused by the operation, the kernel refuses to perform the operation, and returns an error to the user-space code. Ensuring the correctness of the kernel's security checks, then, is a matter of determining which way information can flow between the different objects for each of the operations the kernel supports.

In the virus scanner example, all objects containing private user data are labeled confidential, and the kernel prevents any information flow from confidential to non-confidential objects. Initially, the scanner process is not labeled confidential, and the kernel prevents it from reading the contents of any confidential

object in the system, since doing so would be against the information flow policy specified by the labels. If the scanner changes its label to become confidential, the kernel will allow it to read confidential objects. However, by changing its label to confidential, the scanner also loses its ability to write any data to non-confidential objects, or to change its label back to non-confidential, since threads are only allowed to change their label in a way consistent with information flow restrictions. Thus, regardless of the scanner's actions, the HiStar kernel ensures that confidential data cannot reach non-confidential objects, including the network and the colluding update process.

2.1.1 Levels

HiStar controls information flow by using Asbestos labels, which associate confidentiality *levels* with all objects in the system. For now, let us assume that each object's label specifies just one confidentiality level. The object's level determines how that object's data can be modified or observed. Levels are strictly ordered, and HiStar provides four such levels (**0**, **1**, **2**, and **3**), though this is not an essential feature of our system. Information can flow from objects with lower levels to objects with higher levels, but not the other way around. More precisely, information can flow from an object with level l_1 to an object with level l_2 iff $l_1 \leq l_2$, and for this reason we call the $\leq$ relation *can-flow-to*. This provides the fundamental information flow control mechanism.

It is important to note that the *can-flow-to* relation is transitive. Namely, if information can flow from level l_1 to l_2, and then from l_2 to l_3, then it is also the case that information can flow from l_1 to l_3, since $l_1 \leq l_2$ and $l_2 \leq l_3$ imply $l_1 \leq l_3$. This allows us to reason at a high level about when information can flow between a pair of objects, without having to worry about what intermediate objects the information may go through and how it may do so.

Generally speaking, higher levels can be thought of as representing secret data, because information at a high level cannot flow down to lower levels. Conversely, lower levels can be thought of as representing high-integrity data, because such data can be affected only by other data at a low level. For example, a process at level **1** can only read objects at levels **0** or **1**, and can only modify objects at levels **1**, **2**, or **3**. As a result, objects at level **0** can be thought of as write-protected, and objects with level **2** or **3** as read-protected.

A special fifth level, $\star$ (pronounced "star"), can appear in the label of active "agent" objects, such as threads, which can ask the kernel to perform operations on their behalf. Level $\star$ represents special *downgrading* privilege, and allows a thread to ignore information flow restrictions—in other words, a

Level	Typical meaning by convention
$\star$	has downgrading privileges
0	cannot be written or modified by default
1	default level—no restriction
2	cannot be downgraded or exported by default
3	cannot be read or observed by default

Table 2.1: Conventions for the meaning of different levels in HiStar's Asbestos labels. Only thread and gate objects can have $\star$ in their label.

thread at level $\star$ can read and write objects at any level. In particular, such a thread could read data from an object at level l_1 and write that same data to an object at level l_2, even if $l_1 \not\leq l_2$, thereby *downgrading* data to a lower level and explicitly violating the transitive information flow principle that we have just described. Thus, this privilege level is only given to trusted code, such as the small `wrap` program in the virus scanner example, with the express purpose of downgrading data in a controlled fashion.

It is up to the user-level application code to decide how best to use levels to enforce a particular security policy. In the virus scanner example, confidential and non-confidential objects could be labeled using any pair of levels, as long as the confidential level is greater than the non-confidential one. However, this dissertation always uses levels in a particular fashion, according to the conventions described in Table 2.1.

Although the HiStar kernel tries to enforce information flow control, any realistic implementation is likely going to have covert information flow channels. For example, timing information can often be used by one process to infer whether another process is using up a lot of processor cycles or not, even if the two processes should not be able to communicate. Thus, allowing arbitrary threads in the system to raise their label to the maximum level (**3**) and observe arbitrarily secret information in the system is likely to be a bad idea. To alleviate this problem, HiStar introduces a second label, associated only with active agent objects, such as threads. The *tracking* label of a thread defines what information the thread has potentially observed up to this point, and it is this label that we have been discussing so far. This tracking label is used to determine whether this thread can read or write other objects. A second *clearance* label specifies an upper bound on the thread's *tracking* label, restricting the amount of information the thread can choose to read on its own.

By convention, threads in HiStar have a default tracking label of **1** and a clearance label of **2**. This gives rise to the difference between levels **2** and **3** in the level use conventions: by default, threads cannot arbitrarily raise their label to **3**. A thread with downgrading privilege, or level $*$, can create other objects

and threads with arbitrary labels, and in this way it can specify the information flow policy it wants the kernel to enforce.

2.1.2 Categories

While levels can be used to control information flow, using levels alone does not provide a very flexible security mechanism, as it is impossible to track the flow of multiple kinds of information at the same time—each object has only one level. To allow more complex information flow policies, HiStar provides the notion of an information flow *category*, which allows associating multiple confidentiality levels with each object—one per category. Each category can thus be used to impose different information flow restrictions using the level mechanism. Information can flow between two objects only if the levels of the two objects allow information to flow in every single category.

By using two or more categories, applications can specify and enforce more complex information flow policies than by using levels alone. For example, levels can enforce either secrecy or integrity, but not both at the same time. If an object has level **3**, it may be read-protected, but it can be modified by anyone else in the system. Similarly, if an object has level **0**, it may be write-protected, but anyone can read its contents. We can enforce both secrecy and integrity of an object using two categories, a *read* category and a *write* category. By setting the object's level to **3** in the read category, we can read-protect the object, and by setting the object's level to **0** in the write category, we can write-protect it at the same time.

More precisely, each object has a tracking *label*, which defines the object's confidentiality level in different categories. A label is a function from *categories* to *levels*. Any given label maps all but a small number of categories to some default background level for the object—usually **1** by convention. Thus, a label consists of a default level and a list of categories in which the object has either a higher or lower level than the default. We write labels inside braces, using a comma-separated list of category-level pairs followed by the default level. For example, a typical label might be $L = \{\mathrm{w}\,\mathbf{0},\ \mathrm{r}\,\mathbf{3},\ \mathbf{1}\}$, where w and r are two categories. This is just a more compact way of designating the function

$$L(c) = \begin{cases} \mathbf{0} & \text{if } c = \mathrm{w}, \\ \mathbf{3} & \text{if } c = \mathrm{r}, \\ \mathbf{1} & \text{otherwise.} \end{cases}$$

Each category in which an object's level differs from the default level **1** places a restriction on how other threads may access the object. To see this, consider a thread T with tracking label $L_T = \{\mathbf{1}\}$, and an object O with tracking label $L_O = \{\mathrm{r}\,\mathbf{3},\, \mathbf{1}\}$. Because $L_O(\mathrm{r}) = \mathbf{3} \not\leq \mathbf{1} = L_T(\mathrm{r})$, O has a higher level than T in category r. Hence, no information may flow from O to T, which means the thread cannot read or observe the object. Intuitively, category r is being used to read-protect the object, because the object's label maps category r to level **3**, above the default **1**.

Conversely, an object may have a lower level than the default. If instead an object O' has $L_{O'} = \{\mathrm{w}\,\mathbf{0},\, \mathbf{1}\}$, then $L_T(\mathrm{w}) = \mathbf{1} \not\leq \mathbf{0} = L_{O'}(\mathrm{w})$, and no information can flow from T to O', meaning the thread cannot write to or modify the object. Here, category w is being used to write-protect the object, by mapping category w to level **0** in the object's label, below the default level of **1**. Any given category in an object's tracking label can be used to restrict either reading or writing the object, but not both. While it is, of course, common to restrict both, this requires using two categories.

Like other levels, the special privilege level $\star$ is also scoped to a particular category in a label. Level $\star$ signifies downgrading privileges within a category, and may appear only in the tracking label of threads or gates. Roughly speaking, when a thread is at level $\star$ in a particular category, the kernel ignores that category, but not other categories, in performing label checks for operations by that thread. In other words, if a thread T with tracking label L_T has $L_T(c) = \star$, the thread can bypass information flow restrictions in category c, but not in other categories. We therefore say T *owns* c. A thread that owns a category can also *grant* ownership of the category to other threads using various mechanisms that will be described in the next section.

While there are only a few levels, HiStar supports an effectively unlimited number of categories. Categories are named by 61-bit opaque identifiers, which the kernel generates by encrypting a counter with a block cipher.[1] Encrypting the counter prevents one thread from learning how many categories another thread may have allocated. The counter is sufficiently long that it would take over 60 years to exhaust the identifier space even allocating categories at a rate of one billion per second. Thus, the system permits any thread to allocate arbitrarily many categories.

Any thread can allocate a previously-unused category in HiStar. A thread that allocates a category is granted ownership of that category. We note this is a significant departure from traditional military systems, which use categories but typically support only a fixed number of categories that must be assigned by the privileged security administrator.

[1] The specific length 61 was chosen to fit a category name and a 3-bit level, sufficient to store levels **0** through **3** and $\star$, in the same 64-bit field, which facilitated the label implementation.

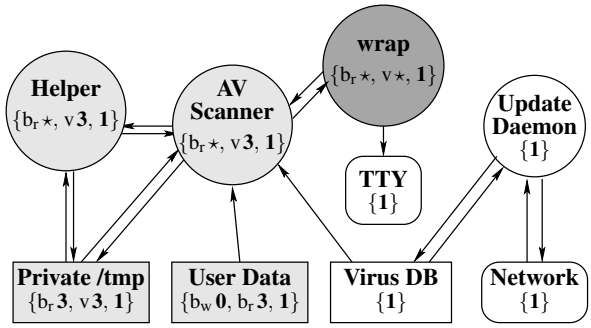

Figure 2.3: Labels on components of the HiStar ClamAV port.

2.1.3 Example

Returning to the virus scanner example, Figure 2.3 shows a simplified version of the tracking labels that would arise if a hypothetical user, "Bob," ran ClamAV on HiStar. Before even launching the virus scanner, permissions must be set to restrict access to Bob's files—otherwise, any process in the system, including the update daemon, could directly read Bob's files and transmit them over the network. This is typically done by the system administrator when initially creating Bob's account, and by Bob when he creates additional files. In Unix, Bob's files would be protected either by setting the file's permission bits to make the file only accessible to the user (mode 0600) or by running the update daemon in a *chroot* jail. In HiStar, labels can achieve equivalent results.

The equivalent of setting Unix permissions bits in HiStar is for Bob to allocate two categories, b_r and b_w, which will be used to restrict read and write access to Bob's files, respectively. Bob labels his data $\{b_r\,\mathbf{3},\ b_w\,\mathbf{0},\ \mathbf{1}\}$. Threads that own b_r can read the data, so ownership of b_r acts like a read capability. Similarly, ownership of b_w acts like a write capability. The authentication mechanism described later on in Section 2.5.2 grants ownership of these two categories to Bob's shell whenever he logs in and provides his password.

The *wrap* program is invoked with all of Bob's privileges, and in particular, with ownership of b_r, the category that restricts read access to Bob's files. *wrap* allocates a new category, v (for **v**irus scanner), to isolate the scanner, and receives sole ownership of this category as a result. *wrap* then creates a private `/tmp` directory writable at level **3** in category v, and launches the scanner labeled **3** in category v. Being labeled with level **3** in category v prevents the scanner, or any process it creates, from communicating to the update daemon or network, except through wrap (which has downgrading privileges in v). Level **3** in

the v category also prevents the scanner, or any program it spawns, from modifying any of Bob's files, because those files all have a lower level (the default level of **1**) in v.

2.1.4 Notation

Almost every operation in HiStar requires the kernel to check whether information can flow between objects. In the absence of level $\star$, information can flow from an object labeled L_1 to one labeled L_2 only if L_2 is at least as high as L_1 in every category. In this case, we say that $L_1 \sqsubseteq L_2$ (pronounced "L_1 can flow to L_2"), or more formally,

$$L_1 \sqsubseteq L_2 \quad \text{iff} \quad \forall c : L_1(c) \leq L_2(c).$$

Level $\star$ complicates matters since it represents ownership and downgrading privileges. A thread T whose tracking label L_T maps a category to level $\star$ can ignore information flow constraints on that category when reading or writing objects. When comparing L_T to an object's tracking label, the $\star$ must be considered either less than or greater than numeric levels, depending on context. When T *reads* an object, $\star$ should be treated as high (greater than any numeric level) to allow observation of arbitrarily secret information. Conversely, when T *writes* an object, $\star$ should be treated as low (less than any numeric level) so that information can flow from T to objects at any level in the category. This shift from high to low implements downgrading—this is the only way for information at a high level to transition to a low level.

Rather than have $\star$ take on two possible values in label comparisons, we use two different symbols to represent ownership, depending on context. The existing $\star$ symbol represents the ownership level of a category when it should be treated low. A new ✪ ("HiStar") symbol represents the same ownership level when it should be treated high. This gives us a notation with six "levels," ordered $\star < \mathbf{0} < \mathbf{1} < \mathbf{2} < \mathbf{3} <$ ✪. However, level ✪ is only used in access rules and never appears in labels of actual objects.

The shifting between levels $\star$ and ✪ required for downgrading is denoted by superscript operators $^{\text{✪}}$ and * that translate $\star$ to ✪ and ✪ to $\star$, respectively. For example, if $L = \{a\star, b\text{✪}, \mathbf{1}\}$, then $L^{\text{✪}} = \{a\text{✪}, b\text{✪}, \mathbf{1}\}$ and $L^{\star} = \{a\star, b\star, \mathbf{1}\}$.

We can now precisely specify the restrictions imposed by HiStar when a thread T labeled L_T attempts to access an object O labeled L_O:

- T can observe O only if $L_O \sqsubseteq L_T^{\text{✪}}$ (i.e., "no read up").
- T can modify O, which in HiStar also implies observing O for practical reasons, only if $L_T \sqsubseteq L_O \sqsubseteq L_T^{\text{✪}}$ (i.e., "no write down").

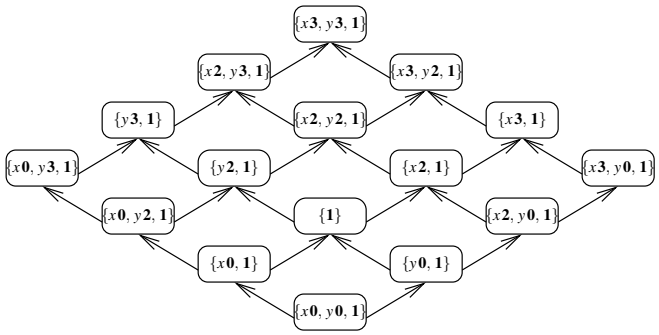

Figure 2.4: Part of the label lattice involving two categories, x and y. All other categories in these labels map to level **1**. Arrows show pairs of labels where the "can flow to" $\sqsubseteq$ relation holds. Information can also flow transitively over multiple arrows at the same time.

These two basic conditions appear repeatedly in our description of the access checks performed by the HiStar kernel.

Labels form a lattice [9] under the partial order of the $\sqsubseteq$ relation. Part of this lattice involving two categories in shown in Figure 2.4. We write $L_1 \sqcup L_2$ to designate the least upper bound of two labels L_1 and L_2. The label $L = L_1 \sqcup L_2$ is given by $L(c) = \max(L_1(c), L_2(c))$. As previously mentioned, threads may choose to raise their label to observe objects with a higher label. To observe an object O labeled L_O, a thread T labeled L_T must raise its tracking label to at least $L'_T = (L_T^{\star} \sqcup L_O)^{\star}$, because that is the lowest label satisfying both $L_T \sqsubseteq L'_T$ and $L_O \sqsubseteq L_T^{\prime\star}$.

2.2 Kernel Interface Design

As previously mentioned, the HiStar kernel is organized around six object types: segments, threads, address spaces, containers, gates, and devices. Through its system call interface, the kernel provides a small number of simple operations on these six object types, with well-defined semantics. Each of these operations correspond to constraints on the labels of the current thread and the objects being accessed. For example, an operation that reads some object O corresponds to the constraint $L_O \sqsubseteq L_T$, where L_O and L_T are the labels of the object and the current thread. These constraints are in turn enforced by the kernel at the system call interface. Because the information flow semantics of all object operations are well-defined, a small and simple kernel can enforce strong information flow control restrictions.

All higher-level abstractions, including the Unix interface, are implemented in untrusted user-level libraries using these six basic kernel object types. As a result, by providing strong information flow control

at the kernel interface, HiStar can also control information flow in complex Unix applications, without having to understand their information flow semantics.

Every HiStar kernel object has a unique, 61-bit *object ID*, a *tracking label*, a *quota* bounding its storage usage, 64 bytes of mutable, user-defined *metadata* (used, for instance, to track modification time), and a few *flags*, such as an *immutable* flag that irrevocably makes the object read-only. Except for threads, objects' tracking labels are specified at creation time and then become immutable. Some objects allow efficient copies to be made with different tracking labels, which is useful in cases when applications want to re-label data.

An object's tracking label controls information flow to and from the object. In particular, the kernel interface was designed to achieve the following property:

> The contents of object A can only affect object B if, for every category c in which A has a higher tracking level than B, a thread owning c takes part in the process.

This is a powerful property. It provides end-to-end guarantees of which system components can affect which others without the need to understand either the components or their interactions with the rest of the system. Instead, to understand what can happen to data labeled with some category c, it suffices to understand and verify only those components that own c.

To revisit the virus scanner example, suppose data from the scanner, labeled $\mathrm{v}\mathbf{3}$, was somehow observed by the update daemon, with a tracking label of $\{\mathbf{1}\}$. It follows that the *wrap* program—the only owner of v—allowed this to happen in some way, either directly or by pre-authorizing actions on its behalf (for instance, by creating a gate). The privacy of the user's data now depends only on the *wrap* program being correct, and not on the virus scanner. In general, we try to structure applications so that key categories are owned by small amounts of code, and hence the bulk of the system is not security-critical.

Although HiStar provides mechanisms that can be used by applications to improve security, it is up to application developers to make use of these mechanisms to secure their applications. Users can also enforce certain security policies on existing applications, without requiring any changes to those applications themselves, by using a small trusted program such as *wrap* to specify an information flow policy. However, all information flow policies enforced by the kernel must come from user-level code; we show in later sections how different applications make use of the kernel's mechanisms to enforce a variety of security policies.

A more technical limitation of HiStar's approach is that it is almost impossible to provide perfect information flow control. Malicious software that is labeled confidential can still leak information through

covert channels—for instance, by modulating CPU usage in a way that affects the response time of other, non-confidential threads. A related problem is preventing malicious software from making even properly labeled copies of data it cannot read. Such copies could divulge unintended information—for instance, allowing someone who just got ownership of a category to read confidential files that were supposed to have been previously deleted. Restricting copies also lets one limit the amount of time malicious software can spend leaking data over covert channels.

To prevent code from accessing or copying inappropriate data, each thread has a *clearance* label, specifying an upper bound both on the thread's own tracking label and on the tracking labels of objects the thread allocates or grants storage to. In the virus scanner example, the update daemon cannot read Bob's private files, which have a tracking label of $\{b_r\,\mathbf{3},\ b_w\,\mathbf{0},\ \mathbf{1}\}$, because the scanner's default clearance label of $\{\mathbf{2}\}$ prevents it from raising its tracking label to level **3** in category b_r.

2.2.1 Persistent Storage

Like any other operating system, HiStar must provide some mechanism for persistent storage on disk. However, any file system provided by HiStar must precisely track information flow, to ensure that it does not violate any information flow control policies. In our virus scanner example, if a malicious virus scanner writes confidential data to the file system, a colluding update process that is not confidential should not be able to read it back from the file system and leak it over the network.

Instead of providing a separate interface to a file system on disk, the HiStar kernel provides a single-level store. This means that the kernel treats all objects, both in memory and on disk, equally. Objects in memory are simply cached versions of objects on disk, and the kernel periodically writes a consistent snapshot of all objects in the system to disk. On bootup, the entire system state is restored from the most recent on-disk snapshot.

A single-level store design simplifies the kernel interface, and reduces the amount of trusted code needed to track information flow on disk as well as in memory. Using a single-level store, HiStar allows the file system to be implemented in an untrusted user-level library using the same kernel abstractions that are used to provide virtual memory. The kernel simply ensures that all information flows according to object labels, both on disk and in memory, and untrusted user space code implements more complex file system semantics, such as directory entries and modification times.

A single-level store also eliminates the need for trusted boot scripts to re-initialize processes such as daemons, which on a more traditional operating systems would not survive a reboot. On the other hand,

persistence opens up a host of other issues, chief among them the fact that one can no longer rely on rebooting to kill off errant applications and reclaim resources.

Indeed, resource exhaustion is a potentially troublesome issue for many systems, including Asbestos. The ability to run a machine out of memory is at best a glaring covert channel and at worst a threat to system integrity. HiStar's single-level store at least reduces the problem to disk-space exhaustion, since all kernel objects are written to disk at each snapshot and can be evicted from memory once stably stored. HiStar prevents disk space exhaustion by enforcing object quotas. Quotas form a hierarchy under top-level control of the system administrator—the only inherent hierarchy in HiStar.

The simplest kernel object type provided by HiStar is a segment, which is a variable-length byte array, similar to a file in other operating systems. The rest of this section discusses other HiStar kernel object types.

2.2.2 Threads

As previously mentioned, each thread T has a tracking label L_T and a clearance label C_T. By default, T has $L_T(c) = \mathbf{1}$ and $C_T(c) = \mathbf{2}$ for most categories c, but the system call

- cat_t *create_category* (void)

chooses a previously unused category, c, by encrypting a counter with a block cipher, and sets $L_T(c) \leftarrow \star$ and $C_T(c) \leftarrow \mathbf{3}$. At that point T is the only thread whose tracking label maps c to a value below the system default of $\mathbf{1}$. In this sense, the label mechanism is egalitarian: no thread has any inherent ownership privileges with respect to categories created by other threads.

T may raise its own tracking label through the system call

- int *self_set_tracking* (label_t L),

which sets $L_T \leftarrow L$ so long as $L_T \sqsubseteq L \sqsubseteq C_T$. This can, for example, let T read a confidential object. T can also lower its clearance in any category (but not below its tracking label), or increase its clearance in categories it owns, using

- int *self_set_clearance* (label_t C),

which sets $C_T \leftarrow C$ so long as $L_T \sqsubseteq C \sqsubseteq (C_T \sqcup L_T^{\star})$.

L_T and C_T restrict the tracking label L of any object T creates to the range $L_T \sqsubseteq L \sqsubseteq C_T$. Similarly, any new thread T' that T spawns must satisfy $L_T \sqsubseteq L_{T'} \sqsubseteq C_{T'} \sqsubseteq C_T$.

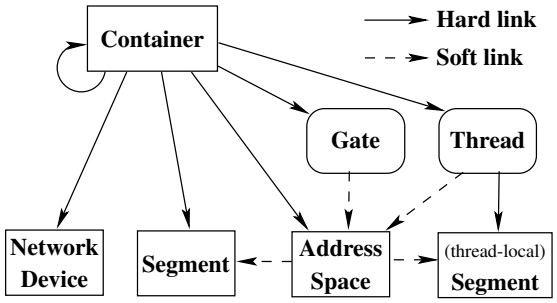

Figure 2.5: Kernel object types in HiStar. A *soft link* names an objects by a particular ⟨container ID, object ID⟩ container entry. Threads and gates, which can own categories (i.e., contain $\star$ in their tracking labels), are represented by rounded rectangles.

2.2.3 Containers

Because HiStar has no notion of superuser yet allows any software to create protection domains, nothing prevents a buggy thread from allocating resources in some new, unobservable, unmodifiable protection domain. Without any other mechanisms, a system administrator would be powerless to reclaim the storage and CPU resources used by such runaway processes, since doing so may violate the information flow control policy specified by the runaway process. However, we must ensure that such resources can nonetheless be deallocated.

HiStar provides hierarchical control over object allocation and deallocation through a *container* abstraction. Containers provide resources to objects, and every HiStar kernel object must exist in some container. Containers can be thought of as Unix directories; like Unix directories, containers hold *hard links* to objects, and some objects may be hard linked to multiple containers. There is a specially-designated root container, which can never be deallocated, and has no parent container—the only such object in the system. Any other object is deallocated once there is no path to it from the root container. Figure 2.5 shows the possible links between containers and other object types.

The container hierarchy separates resource allocation and revocation from all other forms of access control. Even if the label on a particular container prevents a user from accessing it, the user may still be able to reclaim the container's resources by unlinking that container from its parent container. The user-space Unix library places each process in a separate container, making it possible for users to kill runaway processes. The system administrator typically has write access to the root container in the system, providing full control over the system's resources.

When allocating an object, a thread must specify both the container into which to place the object and a 32-byte descriptive string intended to give a rough idea of the object's purpose (much as the Unix *ps* command associates command names with process IDs). For example, to create a container, thread T makes the system call

- id_t *container_create* (id_t D, label_t L, char *$descrip$, int *avoid_types*, uint64_t *quota*).

Here D is the object ID of an existing container, into which the newly created container will be placed. (We use D for containers to avoid confusion with clearance.) L is the desired tracking label for the new container, and *descrip* is the descriptive string. *avoid_types* is a bitmask specifying kernel object types (e.g., threads) that cannot be created in the container or any of its descendants. *quota* is discussed in the next subsection. The system call succeeds only if T can write to D (i.e., $L_T \sqsubseteq L_D \sqsubseteq L_T^\star$) and allocate an object of tracking label L (i.e., $L_T \sqsubseteq L \sqsubseteq C_T$).

Objects can be *unreferenced* from container D by any thread that can write to D. When an object has no more references, the kernel deallocates it. Unreferencing a container causes the kernel to recursively unreference the entire subtree of objects rooted at that container.

HiStar implements directories with containers. By convention, each process knows the container ID of its root directory and can walk the file system by traversing the container hierarchy. The file system uses a separate segment in each directory container to store file names.

A thread T can create a hard link to segment S in container D if it can write D (i.e., $L_T \sqsubseteq L_D \sqsubseteq L_T^\star$) and read S ($L_S \sqsubseteq L_T^\star$). T can thus prolong S's life even without permission to modify S. Any other thread T' must not observe that T has done this, however, unless T could have otherwise communicated to T'—i.e., $L_T \sqsubseteq L_{T'}^\star$ (which need not be the case just because T' has read permission on S).

To achieve this, most system calls specify objects not by ID, but by $\langle$container ID, object ID$\rangle$ pairs, called *container entries*. Container entries ensure that it is safe for the thread to know whether a hard link to some object exists in a particular container, and consequently, whether the object itself still exists or not. For T' to use container entry $\langle D, S\rangle$, D must contain a link to S and T' must be able to read D—i.e., $L_D \sqsubseteq L_{T'}^\star$. Since T had $L_T \sqsubseteq L_D$, this implies that $L_T \sqsubseteq L_{T'}^\star$, and therefore that T is allowed to communicate to T'.

Container entries let the kernel check that a thread has permission to know of an object's existence. When a thread has this permission, it may also read immutable data specified at the object's creation. In particular, for any object $\langle D, O\rangle$, if T can read D, then T can also read O's descriptive string and, unless O is a thread, O's tracking label. (Since thread tracking labels are not immutable, T can only read

the tracking label of another thread T' if $L^{\circledcirc}_{T'} \sqsubseteq L^{\circledcirc}_{T}$.) By examining the tracking labels of objects labeled higher than themselves, threads can determine how they must adjust their own label if they wish to read those objects.

As a special case, every container contains itself. A thread T can access container D as $\langle D, D\rangle$ when $L_D \sqsubseteq L^{\circledcirc}_{T}$, even if T cannot read D's parent, D'. (The root container has a fake parent labeled $\{\mathbf{3}\}$, and must always be referenced this way.) One consequence is that if $L_{D'} \not\sqsubseteq L_D$, a thread with write permission on D' but not D can nonetheless deallocate D in an observable way. By giving D a lower label than its parent in one or more categories, the thread T' that created D effectively pre-authorized a small amount of information to be transmitted from threads that can delete D to threads that can use D. Fortunately, the allocation rules ($L_{T'} \sqsubseteq L_{D'} \sqsubseteq L^{\circledcirc}_{T'}$ and $L_{T'} \sqsubseteq L_D \sqsubseteq C_{T'}$) imply that to create such a D in D', T' must own every category c for which $L_D(c) < L_{D'}(c)$.

2.2.4 Quotas

Every object has a *quota*, which is either a limit on its storage *usage* or the reserved value ∞ (which the root container always has). A container's usage is the sum of the space used by its own data structures and the quotas of all objects it contains. One can adjust quotas with the system call

- int *quota_move* (id_t D, id_t O, int64_t n),

which adds n bytes to both O's quota and D's usage. D must contain O, and the invoking thread T must satisfy $L_T \sqsubseteq L_D \sqsubseteq L^{\circledcirc}_{T}$ and $L_T \sqsubseteq L_O \sqsubseteq C_T$. If $n < 0$, L_T must also satisfy $L_O \sqsubseteq L^{\circledcirc}_{T}$ because the call returns an error when O has fewer than $|n|$ spare bytes, thereby conveying information about O to T.

Threads and segments can both be hard linked into multiple containers; HiStar conservatively "double-charges" for such objects by adding their entire quota to each container's usage. One cannot add a link to an object whose quota may subsequently change. The kernel enforces this with a "fixed-quota" flag on each object. The flag must be set (though a system call) before adding a link to the object, and can never be cleared.

We do not expect users to manage quotas manually, except at the very top of the hierarchy. The system library can manage quotas automatically, though we do not yet enable this feature by default.

2.2.5 Address Spaces

Every thread has an associated address space object containing a list of VA $\rightarrow \langle S, \mathit{offset}, \mathit{npages}, \mathit{flags}\rangle$ mappings. VA is a page-aligned virtual address. $S = \langle D, O\rangle$ is a container entry for a segment to be

mapped at VA. *offset* and *npages* can specify a subset of S to be mapped. *flags* specifies read, write, and execute permissions (and some convenience bits for user-level software).

Each address space A has a tracking label L_A, to which the usual label rules apply. Thread T can modify A only if $L_T \sqsubseteq L_A \sqsubseteq L_T^\star$, and can observe or use A only if $L_A \sqsubseteq L_T^\star$. When launching a new thread, one must specify its address space and entry point. The system call *self_set_as* also allows threads to switch address spaces. When thread T takes a page fault, the kernel looks up the faulting address in T's address space to find a segment $S = \langle D, O \rangle$ and *flags*. If *flags* allows the access mode, the kernel checks that T can read D and O ($L_D \sqsubseteq L_T^\star$ and $L_O \sqsubseteq L_T^\star$). If *flags* includes writing, the kernel additionally checks that T can modify O ($L_T \sqsubseteq L_O$). If no mapping is found or any check fails, the kernel calls up to a user-mode page-fault handler (which by default kills the process). If the page-fault handler cannot be invoked, the thread is halted.

Every thread has a one-page local segment that can be mapped in its address space using a reserved object ID meaning "the current thread's local segment." Thread-local segments are always writable by the current thread. They provide scratch space to use when other parts of the virtual address space may not be writable. For example, when a thread raises its tracking label, it can use the local segment as a temporary stack while creating a copy of its address space with a writable stack and heap.

A system call *thread_alert* allows a thread T' to send an alert to T, which pushes T's registers on an exception stack and vectors T's PC to an alert handler. To succeed, T' must be able to write T's address space A (i.e., $L_{T'} \sqsubseteq L_A \sqsubseteq L_{T'}^\star$) and to observe T (i.e., $L_T \sqsubseteq L_{T'}^\star$). These conditions suffice for T' to gain full control of T by replacing the text segment in A with arbitrary code, as well as for T to communicate information to T'.

2.2.6 Gates

Gates provide protected control transfer, allowing a thread to jump to a pre-defined entry point in another address space with additional privilege. A gate object G has a *gate label* L_G (which may contain $\star$), a *clearance* C_G, a *gate verify label* VL_G, and thread state, including the container entry of an *address space*, an initial *entry point*, an initial *stack pointer*, and some *closure arguments* to pass the entry point function. A thread T' can only allocate a gate G whose label and clearance satisfy $L_{T'} \sqsubseteq L_G \sqsubseteq C_G \sqsubseteq C_{T'}$.

The thread T invoking G must specify a requested label, L_R, and clearance, C_R, to acquire on entry. T also supplies a verify label, L_V, and verify clearance, C_V, to prove ownership and clearance in certain categories without granting those privileges across the gate call. The gate verify label provides a discretionary

limit on what threads can invoke G. Gate invocation is permitted when $L_T \sqsubseteq VL_G$, $L_T \sqsubseteq L_V$, $C_V \sqsubseteq C_T$, and $(L_T^{\circ} \sqcup L_G^{\circ})^{\star} \sqsubseteq L_R \sqsubseteq C_R \sqsubseteq (C_T \sqcup C_G)$. The entry point function can examine L_V and C_V for additional access control. Note that a thread's tracking labels are always explicitly specified by user code, and only verified by the kernel.

Gates are usually used like an RPC service. Unlike typical RPC, where the RPC server provides the resources to handle the request, gates allow the client to donate the initial resources—namely, the thread object that invokes the gate. Arguments and return values are passed across the gate in the thread local segment. Gates can be used to transfer privilege; for example, the login process, described in Section 2.5.2, uses gates to obtain the user's privileges. The use of gates in user-level applications is discussed in more detail in Section 2.4.5.

Gates can also be used to store privilege. As an optimization for this use case, the kernel supports *unbound gates*, which do not have any pre-defined entry point. Instead, the thread that invokes the gate must supply both the address space and entry point that will be executed on gate entry, allowing the thread to execute arbitrary code with the privileges stored in the gate's label and clearance. Unbound gates typically use the gate verify label to restrict the set of threads that can use the gate's privileges.

2.3 Kernel Implementation

Our implementation of HiStar mainly runs on x86-64 processors, such as AMD Opteron and Athlon64 CPUs, although we have ported the code to the 32-bit x86 and SPARC processors as well. The use of a 64-bit processor makes virtual memory an abundant resource, allowing us to make certain simplifications in our design, such as the use of virtual memory for file descriptors, described in the next section.

The single-level store is inspired by XFS [51]. It uses a B+-tree to store an on-disk mapping from object IDs to their location on disk, and two B+-trees to maintain a list of free disk space extents. The first one is indexed by extent size and is used to find appropriately-sized extents, and the other is indexed by extent location and is used to coalesce adjacent extents. Our B+-trees have fixed-size keys and values—object IDs and disk offsets—which significantly simplifies their implementation. Write-ahead logging ensures atomicity and crash-consistency. Disk space allocation is delayed until an object is written to disk, making it easier to allocate contiguous extents.

The kernel performs several key optimizations. It caches the result of comparisons between immutable labels. When switching between similar address spaces, it also invalidates TLB entries with the *invlpg*

instruction instead of flushing the whole TLB by re-loading the page table base register. The *invlpg* optimization makes switching between threads in the same address space efficient: at worst, the kernel invalidates one page translation for the thread-local segment.

2.3.1 Code Size

One of the advantages of HiStar's simple kernel interface is that the fully-trusted kernel can be quite small. Our kernel implementation for the x86-64 processors consists of 18,900 lines of C code (of which 7,300 lines contain a semicolon) and 150 lines of assembly; this is roughly 37% fewer lines of C code than the Asbestos kernel. The source code consists of the following rough components:

- 2,800 lines of architecture-specific code, implementing bootstrapping, virtual memory, and context switching for a specific processor type.
- 4,500 lines of code for B+-trees, write-ahead logging and object persistence.
- 3,800 lines of code for device drivers, including PCI support, DMA-based IDE, console, three timer devices, and four network drivers.
- 7,800 lines of code for system calls, containers, profiling, and other components that are not specific to a particular processor or device.

In comparison, the Asbestos prototype consists of 30,000 lines of code, with 11,100 lines containing a semicolon, not including the trusted microkernel-style file server or the device driver for the Intel e1000 network card.

In all aspects of the design we have tried to optimize for a simpler and cleaner kernel. For example, IPC support, aside from shared memory and gates, is limited to a memory-based futex [13] synchronization primitive, on which the user-level library implements mutexes. The kernel network API consists of three system calls: get the MAC address of the card, provide a transmit or receive packet buffer, and wait for a packet to be received or transmitted. There is no dynamic packet allocation or queuing in the kernel, which simplifies drivers. Our DMA-based Intel eepro100 driver is 500 lines of code, compared to 2,500 in Linux and OpenBSD (not including their in-kernel packet allocation and queuing code). Similarly, our e1000 driver is 450 lines of code, compared to 20,000 lines in the Linux driver. When hardware support for IO virtualization becomes available, we expect to move many device drivers out of the fully-trusted kernel.

To improve the quality of our code, we have tried to use programming language techniques to catch errors; although the success is hard to quantify, it has caught many errors we could have easily missed otherwise. A type-safe enum wrapper helps catch unintended casts by wrapping an *int* value in a C *struct* type. Keeping track of dirty kernel objects is made easier by using the *const* qualifier. Objects are of type *const struct kobject* by default, and only the function *kobject_dirty()* will return a *struct kobject*, after marking the object as being dirty. Compiler warnings and attributes, such as GCC attributes *noreturn* and *warn_unused_result* are used extensively.

2.4 User-level Design

Unix provides a general-purpose computing environment familiar to many people. In designing HiStar's user-level infrastructure, our goal was to provide as similar an environment to Unix as possible except in areas where there were compelling reasons not to—for instance, user authentication, which we redesigned for better security. As a result, porting software to HiStar is relatively straightforward; code that does not interact with security aspects such as user management often requires no modification.

The bulk of the Unix environment is provided by a port of the uClibc library [55] to HiStar. The HiStar platform-specific code is a small layer underneath uClibc that emulates the Linux system call interface, comprising approximately 20,000 lines of code and providing abstractions like file descriptors, processes, fork and exec, file system, and signals. Two additional services—networking and authentication—are provided by separate daemons. A daemon in HiStar is a regular process that creates one or more *service gates* for other processes to communicate with it in an RPC-like fashion.

It is important to note that all of these abstractions are provided at user level, without any special privilege from the kernel. Thus, all information flow, such as the exit status of a child process, is made explicit in the Unix library. A vulnerability in the Unix library, such as a bug in the file system, only compromises threads that trigger the bug—an attacker can only exercise the privileges of the compromised thread, likely causing far less damage than a kernel vulnerability. An untrusted application, such as a virus scanner, can be isolated together with its Unix library, allowing for control over Unix vulnerabilities.

We have ported a number of Unix software packages to HiStar, including GNU coreutils (ls, dd, and so on), ksh, gcc, gdb, perl, Python, ghostscript, the links web browser and OpenSSH. In most cases, porting these software packages required little or no source code modifications, aside from linking it with the uClibc library and our Linux system call emulation layer. The main exception is applications that are concerned with security mechanisms; for instance, we had to modify the OpenSSH login process to grant

user categories instead of changing the Unix user ID of the process upon login. The rest of this section discusses the design and implementation of our Unix emulation library.

2.4.1 File System

The HiStar file system uses segments and containers to implement files and directories, respectively. Each file corresponds to a segment object; to access the file contents, the segment is mapped into the thread's address space, and any reads or writes are translated into memory operations. The implementation coordinates with the user-mode page fault handler to return errors rather than SIGSEGV signals upon invalid read or write requests. A file's length is defined to be the segment's length. Extending a file may require increasing the segment's quota, which is done through a gate call if the enclosing container is not writable in the current context. Additional state, such as the modification time, is stored in the object's metadata.

A directory is a container with a special *directory segment* mapping file names to object IDs. Directory operations are synchronized with a mutex in the directory segment; for example, atomic rename within a directory is implemented by obtaining the directory's mutex lock, modifying the directory segment to reflect the new name, and releasing the lock. Users that cannot write a directory cannot acquire the mutex, but they can still obtain a consistent view of directory segment entries by atomically reading a generation number and busy flag before and after reading each entry. The generation number is incremented by the library on each directory update.

The container ID of the `/` directory is stored by the Unix library in user space and passed to child processes across fork and exec operations. The library also maintains a *mount table segment*, which maps $\langle directory, name \rangle$ pairs onto object IDs. The library overlays mounted objects on directories, much like Unix. Like Plan 9, a process may copy and modify its mount table, for example at user login. The kernel has a *container_get_parent* system call which is used to implement parent directories.

Since file system objects directly correspond to HiStar kernel objects, permissions are specified in terms of labels and are enforced by the kernel, not by the untrusted user-level file system implementation. The tracking label on a file segment is typically $\{r\,\mathbf{3},\ w\,\mathbf{0},\ \mathbf{1}\}$, where categories r and w represent read and write privilege on that file, respectively. Labels are similarly used for directories: read privilege on a directory allows looking up and listing the files in that directory, and write privilege allows creating new files and renaming or deleting existing files.

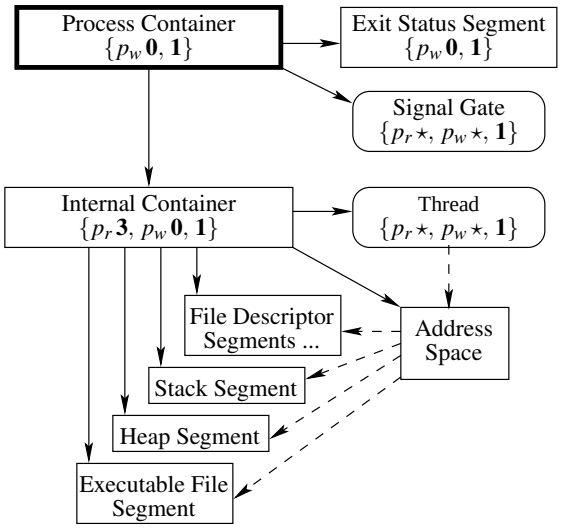

Figure 2.6: Structure of a HiStar process. A process container is represented by a thick border. Not shown are some label components that prevent other users from signaling the process or reading its exit status.

2.4.2 Processes

A process in HiStar is a user-space convention. Figure 2.6 illustrates the kernel objects that make up a typical process; although this may appear complex, it is implemented as untrusted library code that runs only with the privileges of the invoking user.

Each process P has two categories, p_r and p_w, that protect its secrecy and integrity, respectively. Threads in a process typically have a tracking label of $\{p_r \star, p_w \star, \mathbf{1}\}$, granting them full access to the process. The process consists of two containers: a process container and an internal container. The process container exposes objects that define the external interface to the process: a gate for sending signals and a segment to store the process's exit status; not shown is a gate used by gdb for debugging. The process container and exit status segment are labeled $\{p_w \mathbf{0}, \mathbf{1}\}$, allowing read but not write access by threads of other processes (which do not own p_w). The signal gate has a tracking label of $\{p_r \star, p_w \star, \mathbf{1}\}$ and allows other processes to send signals to this process. The internal container, address space, and segment objects are labeled $\{p_r \mathbf{3}, p_w \mathbf{0}, \mathbf{1}\}$, preventing direct access by other processes.

2.4.3 File Descriptors

File descriptors in HiStar are implemented in the user-space Unix library. All of the state typically associated with the file descriptor, such as the current seek position and open flags, is stored in a *file descriptor*

segment. Every file descriptor number corresponds to a specific virtual memory address. When a file descriptor is open in a process, the corresponding file descriptor segment is memory-mapped at the virtual address for that file descriptor number.

Typically each file descriptor segment has a tracking label of $\{f_r\,\mathbf{3},\ f_w\,\mathbf{0},\ \mathbf{1}\}$, where categories f_r and f_w grant read and write access to the file descriptor state. Access to the descriptor can be granted by adding $f_r\star$ and $f_w\star$ to a thread's tracking label. Multiple processes can share file descriptors by mapping the same descriptor segment into their respective address spaces. By convention, every process adds hard links for all of its file descriptor segments to its own container. As a result, ownership of the file descriptor is shared by all processes holding it open, and a shared descriptor segment is only deallocated when it has been closed and unreferenced by every process.

2.4.4 Users

A pair of unique categories u_r and u_w define the read and write privileges of each Unix user u in HiStar, including root. Typically, threads running on behalf of user U have a tracking label containing $u_r\star$, $u_w\star$, and users' private files would have a tracking label of $\{u_r\,\mathbf{3},\ u_w\,\mathbf{0},\ \mathbf{1}\}$. One consequence of this design is that a single process can possess the privilege of multiple users, or perhaps multiple user roles, something hard to implement in Unix. On the other hand, our prototype does not support access control lists. Labels cannot natively express disjunctive access control (either one of these two users should have access to this file), and implementing disjunctions would probably require a gate for every access control group. The authentication service, which verifies user passwords and grants user privileges, is described in more detail in Section 2.5.2.

2.4.5 Gate Calls

Gates provide a mechanism for implementing IPC. As an example, consider a service that generates timestamped signatures on client-provided data; such a service could be used to prove possession of data at a particular time. A HiStar process could provide such a service by creating a *service gate* whose initial entry point is a function that computes a timestamped signature of the input data (from the thread-local segment) and returns the result to the caller. Gates in HiStar have no implicit return mechanism; the caller explicitly creates a *return gate* before invoking the service gate, which allows the calling thread to regain all of the privileges it had prior to calling the service. A fresh category r, which we will call the *return category*, is allocated to prevent arbitrary threads from invoking the return gate. To enforce this, the return

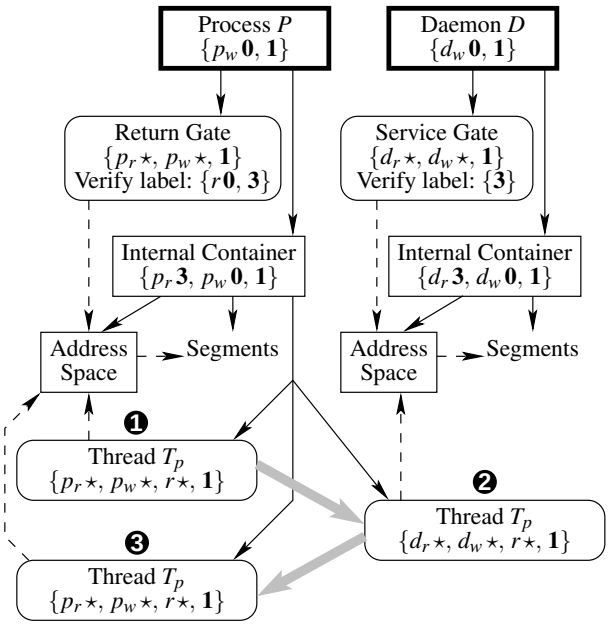

Figure 2.7: Objects involved in a gate call operation. Thick borders represent process containers. r is the return category; d_r and d_w are the process read and write categories for daemon D. Three states of the same thread object T_p, initially part of the P process, are shown: 1) just before calling the service gate, 2) after calling the service gate, and 3) after calling the return gate.

gate's verify label is set to $\{r\,\mathbf{0}, \mathbf{3}\}$, which requires ownership of the return category to invoke it. The caller grants ownership of the return category when invoking the service gate, to allow the service to use the return gate. Figure 2.7 shows such a gate call from process P to daemon D.

Suppose the caller does not trust the signature-generating daemon D to keep the input data private. To ensure privacy, the calling thread can allocate a new taint category t and invoke the service gate with a label of $\{d_r\star, d_w\star, r\star, t\,\mathbf{3}, \mathbf{1}\}$—in other words, tainted in the new category. Ownership of d_r and d_w represents D's privileges to access its own address space and private state, and r represents the privilege to return to the caller process, P. A thread running with this tracking label in D's address space can read any of D's segments, but not modify them (which would violate information flow constraints in category t). However, the tainted thread can make a tainted, and therefore writable, copy of the address space and its segments and continue executing there, effectively *forking* D into an untainted parent daemon and a tainted child in which all of the data is labeled $t\,\mathbf{3}$. The user-space Unix library performs this copy of the address space when it determines that it is running in a read-only address space upon gate entry. Unable to divulge the caller's data, the thread can still compute a signature and return it to the caller. Upon invoking the return gate, the thread regains ownership of category t, allowing it to untaint the computed signature.

Resources for the tainted child copy must be charged against some object's quota. They cannot be charged to D's container, because the thread lacks modification permission when tainted $t\,\mathbf{3}$ (otherwise, it could leak information about the caller's private data to D). Therefore, before invoking the gate, the calling thread creates a container it can use once inside D. In this example, T_p creates a container labeled $\{t\,\mathbf{3}, r\,\mathbf{0}, \mathbf{1}\}$ inside P's internal container.

Forking on tainted gate invocation is not appropriate for every service, because it results in multiple copies of the daemon's address space. Stateless services such as the timestamping daemon are usually well-suited to forking. On the other hand, services that maintain mutable shared state likely want to avoid forking, by refusing tainted gate calls; otherwise, multiple copies of the same mutable shared state will likely lead to incorrect results.

2.4.6 Signals

Signals are implemented by sending an alert to a thread in a process, passing the signal number as an argument to the alert handler. The alert handler invokes the appropriate Unix signal handler for the raised signal. However, sending an alert requires the ability to modify the thread's address space object, which, because of p_w, only other threads in the same process can do. Therefore, to support Unix signals, each

process exposes a *signal gate* in its process container. The gate has a label of $\{p_r\star, p_w\star, \mathbf{1}\}$ and an entry function that sends the appropriate alert to one of the threads in the process, depending on the requested signal number. The verify label on the signal gate is $\{u_w\,\mathbf{0}, \mathbf{3}\}$, where u_w corresponds to the user that is running this process. As a result, only threads that possess the user's privilege can send signals to that user's processes.

2.4.7 Networking

HiStar uses the lwIP [27] protocol stack to provide TCP/IP networking. lwIP runs in a separate *netd* process and exposes a single gate that allows callers to perform socket operations. Operations on socket file descriptors are translated into gate calls to the netd process. By default, netd's process container is mounted as */netd* in mount tables. As an optimization, a process can create a shared memory segment with netd and donate resources for a worker thread to netd. Subsequent netd interactions can then use futexes [13] to communicate over shared memory, avoiding the overhead of gate calls.

The network device is typically labeled $\{n_r\,\mathbf{3}, n_w\,\mathbf{0}, i\,\mathbf{2}, \mathbf{1}\}$, where n_r and n_w are owned by netd, and i taints all data read from the network. Because netd cannot bypass the tainting with i or leak tainted data in other categories, it is mostly untrusted. A compromised netd can only mount the equivalent of a network eavesdropping or packet tampering attack.

2.4.8 Explicit Information Leaks

The Unix interface was not designed to control information flow. As a result, emulating certain aspects of Unix on HiStar requires information leaks. HiStar implements these leaks at user level, through explicit *untainting gates*. By convention, when spawning a tainted thread, or tainting a thread through a gate call, the user-space library supplies the tainted thread with the container entry of an untainting gate. The new thread can invoke this gate to leak certain kinds of information, such as the fact it is about to exit (so the parent shell can reclaim resources and return to the command prompt). Not all categories have untainting gates; whether or not to create one is up to the category's owner.

Currently our Unix library provides untainting gates for up to three operations: process exit, quota adjustment, and file creation. Process exit reports the exit status of a tainted process to its untainted parent. Quota adjustment allows a tainted process to obtain more quota (or return a part of its quota) to its untainted parent container. File creation allows a tainted process to create a new tainted file in an untainted directory. Of these three operations, file creation provides by far the biggest information flow,

declassifying the name of the newly created file. Low-secrecy applications concerned only with accidental disclosure allow of these operations for their categories. Higher-secrecy applications may choose to set fixed quotas for tainted objects and only declassify process exits. Moderate-secrecy applications may want to prohibit just file creation. The next section shows examples of such applications.

2.5 Applications

The Unix environment described in the previous section allows for general-purpose computing on HiStar, but does not provide any functionality qualitatively different from Linux. HiStar's key advantage is that it enables novel, high-security applications to run alongside a familiar Unix environment. This section presents some applications that take advantage of HiStar to provide security guarantees not achievable on typical Unix systems.

2.5.1 Anti-Virus Software

We have implemented an untrusted virus scanner, as suggested in several examples, by porting the ClamAV scanner [8] to HiStar and using the *wrap* program to run it in isolation. To provide strong isolation, *wrap* does not create the standard Unix untainting gates for category *v*. *wrap* also limits the amount of data that can be leaked through covert channels by killing ClamAV after some period of time. This amount of time is fixed in our current prototype, but could be scaled with the amount of work that ClamAV is expected to perform.

ClamAV and its database must be periodically updated to keep up with new viruses. In HiStar, the update process runs with the privilege to write the ClamAV executable and virus database; however, it cannot access private user data. Even if a compromised update installs arbitrary code in place of ClamAV, the tracking label set by *wrap* when running ClamAV ensures that private information cannot be exported.

2.5.2 User Authentication

User authentication provides a good example of how HiStar can minimize trusted code. Most operating systems require a highly-trusted process to validate authentication requests and grant credentials. For example, the Unix login program runs as superuser to set the appropriate user and group IDs after checking passwords. Even a privilege-separated server such as OpenSSH requires a superuser component to be able to launch shells for successfully authenticated users.

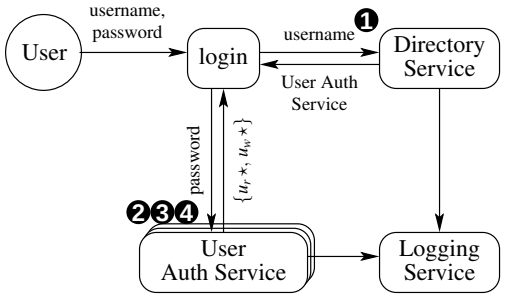

Figure 2.8: A high-level overview of the authentication system.

In contrast, HiStar authenticates users without any highly-trusted processes, and allows users to supply their own authentication services. Even if a user accidentally provides his or her password to a malicious authentication service, HiStar ensures that only one bit of information about the user's password—whether the authentication succeeded using that password—is leaked. Providing such isolation under a traditional operating system would be difficult.

Figure 2.8 shows an overview of the HiStar authentication facility. Logically, four entities coordinate to authenticate a user: a login client, a directory service, a per-user authentication service, and a logging service. Of these, the logging service is simplest; the directory and user authentication services trust it to maintain an append-only log, while it trusts them not to exhaust space with spurious entries.

The login client initiates authentication. It typically consists of an instance of the web server or *sshd* that knows a username and password and wishes to gain ownership of the user's read and write categories, u_r and u_w. Login minimally trusts the directory to interpret the username properly (without which authentication could fail or return the wrong credentials). However, login does not trust the other components, and importantly does not trust anyone with the user's password. Conversely, no other component trusts login until it authenticates itself.

The directory service maintains a list of user accounts. Its job is to map usernames to user authentication service daemons. Login begins the authentication process by asking the directory for a particular username. The directory responds with the container entry of a gate to the user's authentication service. The directory is controlled by the system administrator, but is untrusted except minimally by login and the logger as described above.

Each user runs an authentication service daemon that owns u_r and u_w; the daemon's job is to grant those categories to login clients that successfully authenticate themselves. Conceptually, this is simple: login sends the password to the authentication service, which checks it and, if correct, grants u_r and u_w

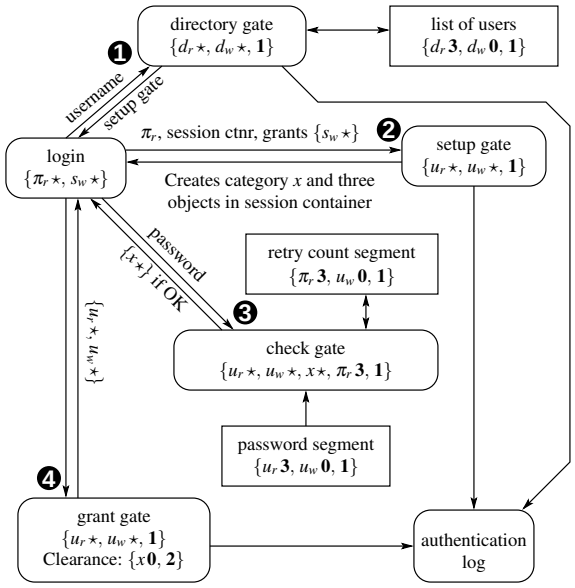

Figure 2.9: A detailed view of the interactions between authentication system components. The setup gate, check gate and grant gate (2, 3 and 4) are all part of the user's authentication service.

back to login. Since the authentication service is under the user's control, it can, at the user's option, support non-password techniques such challenge-response authentication.

The complication is that login does not trust the authentication service with the user's password. After all, a mistyped username or malicious directory could connect login to the wrong authentication service. Even the right service might be compromised, which should reveal only the user's password hash, not his password. With challenge-response authentication, a similar man-in-the-middle threat exists. The solution is for login to invoke the authentication service three times: first to set things up, second to check the password, and third to finally gain ownership of u_r and u_w. The second step runs tainted, thereby protecting the secrecy of the password.

Figure 2.9 shows the authentication sequence in more detail. In Step 1, login learns of the appropriate user's setup gate from the directory service. Then it allocates two categories: π_r, the password read (secrecy) category, protects the password from disclosure. The s_w category controls write access to a *login session container*, which login creates with label $\{s_w \mathbf{0}, \mathbf{1}\}$.

In Step 2, login invokes the user's setup gate, granting the user's code $s_w \star$. The setup gate logs the authentication attempt and allocates a new category, x, to be granted to login after successful authentication. Before returning, the setup gate code (together with login, as we will discuss later) creates three objects in

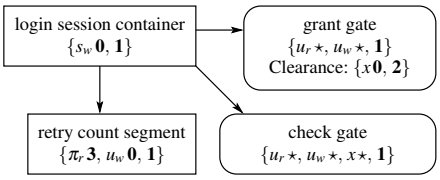

Figure 2.10: Objects created by the user's setup gate in the session container.

the session container, shown in Figure 2.10. The first is a *retry count segment*, used to bound the number of password guesses per logged invocation of the setup gate. The second is an ephemeral *check gate*, used to check passwords while tainted; its closure arguments specify the object ID of the retry count segment. The third is an ephemeral *grant gate* with clearance $\{x\,\mathbf{0},\,\mathbf{2}\}$.

In Step 3, login calls the check gate with the password, tainting the thread $\pi_r\,\mathbf{3}$. If the password is correct and the retry count okay, the gate code grants x back to login. (Optionally, the check gate may accept a verify label of $\{root_w\,\mathbf{0},\,\mathbf{3}\}$ instead of a password, to emulate a Unix users' trust of root.) Once login owns x, it calls the grant gate in Step 4 to obtain u_r and u_w. The grant gate logs the authentication success before returning, which is why it must be separate from the tainted check gate, which cannot talk to the logging service.

In Step 2, creating the retry count segment, which is labeled $\{\pi_r\,\mathbf{3},\,u_w\,\mathbf{0},\,\mathbf{1}\}$, requires combining the privileges of two mutually-distrustful entities: login, with a clearance of $\pi_r\,\mathbf{3}$, and the user's code, with a label of $u_w\star$. The user's code will not grant $u_w\star$ to login before a successful authentication. Similarly, login does not trust the user's setup gate code with a clearance of $\pi_r\,\mathbf{3}$.

To see why login cannot invoke the setup gate with a clearance of $\pi_r\,\mathbf{3}$, consider what malicious setup gate code can do given such a clearance: It can create a long-lived segment S labeled $\{\pi_r\,\mathbf{3},\,u_r\,\mathbf{3},\,\mathbf{1}\}$, and a long-lived thread T labeled $\{\pi_r\,\mathbf{3},\,u_r\star,\,\mathbf{1}\}$. Both can be in a container inaccessible to login. The setup code can furthermore point the check gate to a "trojaned" variant of the password checker that writes the password to S. Finally, T can read S and leak the password through a covert channel over a long period of time. T and S will persist long after login has destroyed all objects it knows about with a clearance of $\pi_r\,\mathbf{3}$.

To solve this problem, the developers of the user's authentication service and the login client agree ahead of time on a function that both of them want to execute to create the retry count segment. Then, before invoking the setup gate, login creates a code segment containing the code of the previously agreed-upon function, as well as a gate G that invokes this code with a clearance of $\pi_r\,\mathbf{3}$. Additionally, login marks the code segment and address space objects invoked by G as *immutable* in the kernel. Because

these objects are immutable, the user's setup gate code can verify their contents and be assured that invoking G with $u_w\star$ will execute only the agreed upon code and not somehow result in login usurping ownership of u_w. In this manner, two mutually-distrustful parties can safely execute mutually agreed-upon code with their combined privilege.

The authentication service implementation is fairly small. The logging service consists of 58 lines of code; the directory service consists of 188 lines, and the standard password-based user authentication service consists of 233 lines of code. Common library code that allows combining privileges to create the retry count segment is 370 lines of C++ code, and the mutually agreed-upon code to create the retry count segment is 30 lines of assembly. Aside from security, another advantage of privilege-separating authentication is that the processes can keep relatively small labels, improving the performance of label operations.

2.5.3 VPN Isolation

Many networks rely so heavily on firewalls for security that the prospect of bridging them to the open Internet poses a serious danger. Indeed, this is how the Slammer worm disabled a safety monitoring system at a nuclear power plant in 2003 [41]. At the same time, it has become quite common for people to connect home machines and laptops to otherwise firewalled networks through encrypted virtual private networks (VPNs). When VPNs let the same machine connect to either side of a firewall, they risk having malware either infect internal machines or (as the Sircam worm did) divulge sensitive documents to the world.

In HiStar, however, one can track the provenance of data with labels and precisely control what flows between networks. The bootstrap procedure already labels the network device to taint anything received from the Internet $\{i\mathbf{2}, \mathbf{1}\}$ and block from transmission anything more tainted. One can analogously label all VPN input $\{v\mathbf{2}, \mathbf{1}\}$ and block any more tainted VPN output. Such a configuration completely isolates the two networks from each other except as specifically permitted by the owners of i and v. For example, users might be allowed to untaint i (meaning import external data) when the file passes a virus checker, such as the one in Section 2.5.1.

We have implemented VPN isolation around the popular OpenVPN package [37]. Figure 2.11 shows the components of the system and their tracking labels: The VPN runs a second lwIP stack which talks to the OpenVPN client over a *tun* device. Porting OpenVPN to HiStar required implementing a tun character device in the file system library (200 lines of code) and a tun "device driver" for lwIP (100 lines of code).

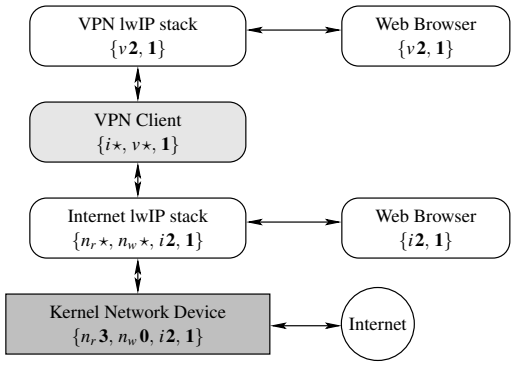

Figure 2.11: Secure VPN application. The VPN client is trusted to label incoming VPN packets with $\{v\mathbf{2}\}$, reject any outgoing packets tainted in category i, and properly encrypt/decrypt data. The kernel network device is completely trusted. Neither of the lwIP stacks is trusted.

OpenVPN swaps between v and i taints on the data it encrypts. Users select which network to use by mounting the appropriate lwIP process on */netd* (much like Plan 9). Not shown are untainting gates, which for this application allow processes to leak exit, quota, and file creation events, as discussed in Section 2.4.8.

VPN isolation is interesting because it applies a broad policy potentially affecting most processes in the system, yet requires only a localized change. This would be difficult to achieve in a capability-based system, for instance.

2.5.4 Web Server

The original motivating application for Asbestos [10] was its web server, which isolated different user's data to tolerate buggy or malicious web service code. We have built a similar web server for HiStar, with a few differences. HiStar's connection demultiplexer controls resources granted to each worker daemon through containers, allowing control over resources used by untrusted code. Authentication uses an instance of the daemon described in Section 2.5.2. The SSL code is untrusted, and does not even have access to the SSL certificate private key. Figure 2.12 shows the overall architecture of this privilege-separated SSL web server. The web server is built from a number of mutually-distrustful components to reduce the effects of the compromise of any single component. We first describe how requests are handled in this web server, and then describe the security properties achieved by this server in the next subsection.

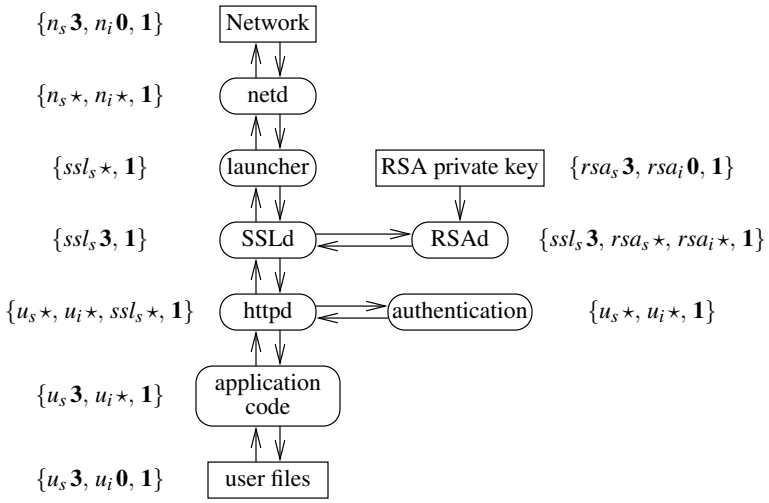

Figure 2.12: Architecture of the privilege-separated SSL web server running on HiStar. Each process, indicated by a rounded box, is largely distrustful of other components. Rectangles represent devices and files. The HiStar tracking label of each component is also shown. A fresh ssl_s category is allocated for each connection, while other categories are long-lived.

The TCP/IP stack is implemented by the user-space ***netd*** process, which has direct access to the kernel network device. netd provides a traditional sockets interface to other applications on the system, which is used by our web server to access the network.

User connections are initially handled by the ***launcher*** process, which listens for incoming connections from web browsers and allocates resources to handle each request. For each request, the launcher spawns an ***SSLd*** daemon to handle the SSL connection with the user's web browser, and an ***httpd*** daemon to process the user's plaintext HTTP request. The launcher then proxies data between SSLd and the TCP connection to the user's browser. SSLd, in turn, uses the ***RSAd*** daemon to establish an SSL session key with the web browser, by generating an RSA signature using the SSL certificate private key kept by RSAd.

The httpd process receives the user's decrypted HTTP request from SSLd and extracts from it the user's password and request path. It then authenticates the user, by sending the password from the HTTP request to that user's ***password checking agent*** from the HiStar authentication service. If the authentication succeeds, httpd receives ownership of the user's secrecy and integrity categories, and executes the appropriate ***application code*** with the user's privileges (in our case, we run GNU ghostscript to generate a PDF document). Any output from the application is sent back to the user's web browser, via SSLd for encryption.

Component	Lines of Code	Tracking Label	Ownership Label	Effects of Compromise
netd	350,000	$\{\}$	$\{n_s, n_i\}$	Same as an active network attacker, observing and injecting traffic
launcher	310	$\{\}$	$\{ssl_s\}$	Obtain plaintext of user requests, including passwords, and server responses
SSLd	340,000	$\{ssl_s\}$	$\{\}$	Corrupt request or response, or send unencrypted data to same user's browser
RSAd	4,600	$\{ssl_s\}$	$\{rsa_s, rsa_i\}$	Disclose the server's SSL certificate private key
httpd	300	$\{\}$	$\{u_s, u_i, ssl_s\}$	Full access to data in attacker's account, but not to other users' data
authentication	320	$\{\}$	$\{u_s, u_i\}$	Full access to data of the user whose agent is compromised, but no password disclosure
application	680,000+	$\{u_s\}$	$\{u_i\}$	Send garbage (but only to same user's browser), corrupt user data (for write requests)

Table 2.2: Components of the HiStar web server, their complexity measured in lines of C code, their tracking and ownership labels, and the worst-case results of an attacker exploiting a vulnerability in that component. Not included in the lines of code are shared libraries such as libc.

2.5.5 Web Server Security

The HiStar web server architecture has no hierarchy of privileges, and no fully trusted components; instead, most components are mutually distrustful, and the effects of a compromise are typically limited to one user, usually the attacker himself. Table 2.2 summarizes the security properties of this web server, including the complexity of different components and effects of compromise.

The largest components in the web server, SSLd and the application code, are minimally trusted, and cannot disclose one user's private data to another user, even if they are malicious. The application code is confined by the user's secrecy category, u_s, and it is httpd's job to ensure that the application code is labeled with u_s when httpd runs it. Although the application code owns the user's integrity category, u_i, this only gives it the privilege to write to that user's files, but not to export them.

SSLd is confined by the ssl_s secrecy category, which is a fresh category allocated by the launcher for each new connection. Both the launcher and httpd own ssl_s, allowing them to freely handle encrypted and decrypted SSL data, respectively. However, SSLd can only communicate with the user's web browser, via the launcher, or with httpd.

SSLd is also not trusted to handle the SSL certificate private key. Instead, a separate and much smaller daemon, RSAd, has access to the private key, and only provides an interface to generate RSA signatures for SSL session key establishment. Not shown in the diagram is a category owned by SSLd that

allows it and only it to invoke RSAd. Although a compromised RSAd can expose the server's SSL private key, it cannot directly compromise the privacy of user data, because RSAd runs confined with each user connection's ssl_s category.

Side-channel attacks, such as [1], might allow recovery of the private key; OpenSSL enables RSA blinding by default to defeat timing attacks such as [5]. To prevent an attacker from observing intermediate states of CPU caches while handling the private key, RSAd starts RSA operations at the beginning of a 10 msec scheduler quantum (each 1024-bit RSA operation takes 1 msec), and flushes CPU caches when context switching to or from RSAd (with support from the kernel), at a minimal cost to overall performance.

The HiStar authentication service used by httpd to authenticate users is the same exact service that was described in Section 2.5.2, which further reduces the effects of compromise to a single user, and ensures that even in the case of a malicious authentication service, the user's password is not disclosed.

In our current prototype, httpd always grants ownership of u_i to the application code, giving it write access to user data. A conservative implementation may want to avoid granting u_i to code that performs read-only requests, to avoid user data corruption due to buggy read request code.

Our web server does not use SSL client authentication in SSLd. Doing so would require either trusting all of SSLd to authenticate all users, or extracting the client authentication code into a separate, smaller trusted component. In comparison, the password checking agent in the HiStar authentication service is 320 lines of code.

One caveat of our current implementation is the absence of SSL session caching. Because a separate instance of SSLd is used for each client request, clients cannot reuse existing session keys when connecting multiple times, requiring public key cryptography to establish a new session key. This limitation can be addressed by adding a trusted SSL session cache that runs in a different, persistent process, at the cost of increasing the amount of trusted code.

2.6 Performance

To evaluate the performance implications of HiStar's architecture, we compared it to Linux and OpenBSD under several benchmarks. The benchmarks ran on three identical systems, each with a 2.4 GHz AMD Athlon64 3400+ processor, 1GB of main memory, and a 40 GB, 7,200 RPM Seagate ST340014A EIDE hard drive. The first machine ran HiStar; the second ran Fedora Core 5 Linux with kernel version 2.6.16-1.2080_FC5 x86_64 and an ext3 file system; the third ran 32-bit OpenBSD 3.9 i386 with an in-memory *mfs*

Benchmark	HiStar	Linux	OpenBSD
IPC benchmark, per RTT	3.11 μsec	4.32 μsec	2.13 μsec
Fork/exec, per iteration	1.35 msec	0.18 msec	0.18 msec
Fork/exec, dynamic linking	—	0.45 msec	0.38 msec
Spawn, per iteration	0.47 msec	—	—
LFS small, create, async	0.31 sec	0.316 sec	0.22 sec
... per-file sync	459 sec	558 sec	—
... group sync	2.57 sec	—	—
LFS small, read, cached	0.16 sec	0.068 sec	0.14 sec
... uncached	6.49 sec	1.86 sec	—
... no IDE disk prefetch	86.4 sec	86.6 sec	—
LFS small, unlink, async	0.090 sec	0.244 sec	0.068 sec
... per-file sync	456 sec	173 sec	—
... group sync	0.38 sec	—	—
LFS large, sequential write	2.14 sec	3.88 sec	—
... sync random write	93.0 sec	89.7 sec	—
LFS large, uncached read	1.96 sec	1.80 sec	—

Table 2.3: Microbenchmark results on HiStar, Linux and OpenBSD.

file system—a 64-bit version of OpenBSD 3.8 for amd64 performed strictly worse in every benchmark. We did not run synchronous file system benchmarks under OpenBSD, because we could not disable IDE write caching.

2.6.1 Microbenchmarks

To evaluate the performance of specific aspects of HiStar, we chose four microbenchmarks: LFS small-file and large-file benchmarks [43], an IPC benchmark which measures the latency of communication over a Unix pipe, and a fork/exec benchmark that measures the latency of executing `/bin/true` using fork and exec. All microbenchmarks and `/bin/true` were compiled statically to eliminate dynamic linking overhead. Table 2.3 shows the performance of the four microbenchmarks on three different operating systems.

For the IPC benchmark, two processes are created, connected by two uni-directional pipes; each process sends any messages it receives back to the other process. The benchmark measures the average round-trip time taken to transmit an 8-byte message, over one million round-trips. HiStar performs better than Linux in this benchmark, but somewhat slower than OpenBSD.

HiStar's performance noticeably suffers in the fork and exec microbenchmark. In part, this is because Linux and OpenBSD pre-zero memory pages, which HiStar does not yet do. Moreover, while OpenBSD

and Linux require 9 system calls to fork a child, have the child execute `/bin/true`, have `/bin/true` exit, and have the parent wait for the child, the same workload requires 317 system calls on top of HiStar's lower-level interface. However, the flexibility provided by a lower-level interface allows us to implement more efficient library calls, such as *spawn*, which directly starts a new process running a specified executable. The *spawn* function runs 3 times faster than the equivalent fork and exec combination, issuing only 127 system calls per iteration. We note that use of dynamic linking would reduce the relative performance difference between HiStar and Linux.

The LFS small file benchmark creates, reads, and unlinks 10,000 1kB-sized files and reports the total running time for each of these three phases. We measured different variations of the phases, as shown in Table 2.3. The asynchronous and cached variations show HiStar has comparable performance to the other systems for requests that go to cache. The uncached read phase measures the time to read 10,000 small files from disk. Here Linux significantly outperforms HiStar, averaging less than $1/10$th the disk's 8.3 msec rotational latency to read each file. We attribute this performance to read look-ahead in the IDE disk [46], because Linux clusters files from the same directory while HiStar does not. Disabling lookahead, HiStar and Linux perform comparably.

In the synchronous unlink phase, HiStar performs significantly worse than Linux. This is because we implement *fsync* of a directory by checkpointing the entire system state to disk, whereas Linux only writes out the modified directory entry. Synchronous file creation in HiStar also checkpoints the entire system state; however, its performance is comparable to Linux because ext3 performs more writes in this case. Write-ahead logging allows HiStar to achieve acceptable *fsync* performance by queuing updates in a sequential on-disk log. Logged updates are applied in batches; during each run of the synchronous small file benchmarks, the contents of the on-disk log were applied to disk about 10 times (once for approximately every 1,000 synchronous operations).

The single-level store offers a new *group sync* consistency choice not possible under Linux. In group sync, the system state is checkpointed to disk only once at the end of each benchmark phase. The single-level store guarantees that the application either runs to completion or appears never to have started. Using group sync in HiStar, some applications may achieve a significant speedup over Linux, as high as a factor of 200 for applications similar to the LFS small file benchmark.

For the LFS large file benchmark, we evaluated three phases. In the first phase, a 100MB file was created by sequentially writing 8KB chunks, with a single call to *fsync* at the end of the phase. HiStar achieves close to the maximum disk bandwidth of 58MB/sec [46]; we suspect that block-based (rather than extent-based) allocation in ext3 accounts for Linux's slightly lower performance.

Benchmark	HiStar	Linux	OpenBSD
Building HiStar kernel	6.2 sec	4.7 sec	6.0 sec
Transferring 100MB with wget	9.1 sec	9.0 sec	9.0 sec
Virus-checking a 100MB file	18.7 sec	18.7 sec	21.2 sec
. . . with isolation wrapper	18.7 sec	—	—

Table 2.4: Application-level benchmark results.

The second phase tested random write throughput; 100MB worth of 8KB chunks were written to random locations in the existing file, and the modifications were *fsync*ed to disk for each 8KB write. In the case of pre-existing segments, HiStar allows modified segment pages to be flushed to disk (modified in-place) without checkpointing the entire system state. As a result, the performance is again quite close to that of Linux, since each random write involves flushing two 4KB pages to disk both in Linux and in HiStar.

The third phase of the large-file benchmark tested read performance by sequentially reading the 100MB file in 8KB chunks. The performance is approximately the same between HiStar and Linux. Currently the HiStar prototype does not support paging in of partial segments, so the entire 100MB file segment is paged in when the file is first accessed—a limitation we plan to address in the future. As a result, the performance of random reads differs little from the sequential case.

2.6.2 Application Performance

For an application-level benchmark, we built the HiStar kernel using GNU make 3.80 and GCC 3.4.5 on the three operating systems; Table 2.4 summarizes the results. HiStar is somewhat slower than Linux and comparable to OpenBSD. In HiStar, most of the CPU time in this benchmark is spent in user space. Since most of our optimization efforts to date have focused on the kernel, we expect HiStar to improve on this benchmark as we move to optimizing the Unix library.

HiStar also achieves good network throughput. When downloading a 100MB file using *wget*, the results show all three operating systems could saturate a 100Mbps Ethernet. Finally, we measured the time taken to check a 100MB file containing randomized binary data for viruses using ClamAV; HiStar performs competitively with Linux and OpenBSD, both with and without the use of the wrapper described in Section 2.5.1.

Chapter 3

Distributed Information Flow Control

Large-scale applications, such as web servers, often require the resources of multiple physical machines for performance scalability. Although HiStar can provide strong security guarantees on a single machine, it falls short of enforcing an overall security policy for such distributed applications. This chapter presents DStar, a system for enforcing an overall information flow control policy across machines in a distributed system. DStar allows one to reason about information flow in a distributed system—namely, which messages sent *to* a machine can affect which messages sent *from* the machine—all without trusting or even understanding most of the code in the system.

Superficially, DStar can be thought of as a way to extend HiStar's information flow control mechanisms over the network. However, controlling information flow in a distributed setting brings with it a number of additional challenges. While HiStar has the luxury of an entirely trusted kernel, in the distributed setting there may not be any fully trusted code on any machine. For example, some web server's front-end machines may use SSL private keys, some back-end machines may store orders and credit card numbers, but no single machine needs access to all this data. Moreover, the kernel on a single machine has access to all information necessary to make access control decisions. In a distributed setting, one machine may need authorization from a second machine to disclose data to a third machine. Even attempting to obtain such authorization can inappropriately leak information.

Another, more subtle, complication of the distributed setting is resource allocation, a common source of covert communication channels, particularly when crossing protection domains. Within a single machine, HiStar solves this problem by donating resources such as memory across inter-domain calls, thereby saving the callee from having to commit any of its own resources in a way that would be detectable

by other callers. In a distributed environment, resources on one machine cannot be donated to processes on a different machine, yet receiver buffer space should still not be implicitly granted to senders.

Each DStar machine runs an *exporter* daemon that allows processes to communicate over the network while preserving information flow constraints. At a high level, exporters exchange the local operating system's information flow restrictions for authenticated encryption of network messages and vice versa. Thus, while confined processes cannot directly access the network, exporters provide them an interface for safe communication with other processes on different machines. The job of the exporter, then, is to decide when it is safe to send or receive a particular network message.

The main idea behind DStar is the use of self-certifying *categories* to specify information flow restrictions. Each self-certifying category includes the public key of its creator in the category name. This allows any exporter to determine who is trusted to maintain information flow restrictions for that category by verifying certificate signatures starting with the creator's public key, without involving a trusted central authority by design.

DStar exporters use HiStar to confine untrusted code, but DStar does not mandate any particular confinement mechanism. A Unix port of DStar permits partially trusted, unconfined code on legacy operating systems to inter-operate with confined code on HiStar machines. This facilitates incremental deployments that migrate only the riskiest parts of the system to HiStar. Capability-based operating systems could also be used by DStar to isolate code running on a machine, with some limitations we discuss later. Distributed capability systems, on the other hand, cannot provide confinement of untrusted code.

DStar exporters are mutually distrustful, and are not trusted by the underlying operating system. In addition to improved security, a decentralized design makes it easier to integrate or communicate between administrative domains, because there are no central databases to merge. Avoiding a centralized design also ensures that there are no inherent centralized bottlenecks that would impede scalability.

Using DStar, we show how to build a highly privilege-separated, scalable, distributed web server that provides strong security guarantees by restricting information flow. For instance, the SSL certificate private key is not readable even by the SSL library, and can only be used for legitimate SSL negotiation. Authentication tokens such as passwords are protected, so that even the bulk of the authentication code cannot disclose them. Private user data returned from a storage server can only be written to an SSL connection over which the appropriate user has authenticated himself. Even fully-compromised server machines can only subvert the security of users that use or had recently used those machines.

HiStar Tracking Label	DStar Tracking Label	DStar Ownership Label
$\{\mathbf{1}\}$	$\{\}$	$\{\}$
$\{u_s\,\mathbf{3}, \mathbf{1}\}$	$\{u_s\}$	$\{\}$
$\{u_i\,\mathbf{0}, u_s\,\mathbf{3}, \mathbf{1}\}$	$\{u_i, u_s\}$	$\{\}$
$\{u_i\star, u_s\star, \mathbf{1}\}$	$\{\}$	$\{u_i, u_s\}$
$\{u_i\star, u_s\,\mathbf{3}, \mathbf{1}\}$	$\{u_s\}$	$\{u_i\}$

Table 3.1: Equivalent HiStar and DStar labels. Because DStar categories are typed, a *s* subscript indicates a secrecy category, and an *i* subscript indicates an integrity category.

3.1 Design

DStar represents information flow restrictions using a simplified version of HiStar labels that are easier to reason about in distributed systems without global knowledge or authority. All DStar categories are one of two types, *secrecy* or *integrity*, determined at category creation time. Secrecy categories restrict who can observe and disseminate the data, and correspond to categories typically mapped to level **3** in a HiStar label. Conversely, integrity categories constrain who can modify the data or vouch for its integrity, and correspond to categories typically mapped to level **0** in HiStar. We use the suffix *s* and *i* to indicate whether a category is a secrecy category or an integrity category, respectively. DStar has no equivalent of level **2**.[1] All data in the system has a tracking label, which is a set of categories restricting who can observe or modify the data. There are no levels in DStar labels, because category types implicitly specify the equivalent level. Bypassing any category's restriction requires privilege in that category; in HiStar, this corresponds to level $\star$, but DStar represents this privilege with a separate *ownership* label. Table 3.1 shows a few examples of equivalent HiStar and DStar labels.

Every message M sent over the network in the DStar protocol has a tracking label L_M specifying how its contents have been and should be protected by hosts' local operating systems. Every process P that can send or receive messages also has a tracking label, L_P, which determines what messages P can send or receive.

Intuitively, P cannot receive any message whose tracking label contains any secrecy category not in L_P, or any message whose tracking label is missing any integrity category present in L_P. Conversely, if P wants to send a message, the tracking label of the message must contain at least all of the secrecy categories in L_P, and at most all of the integrity categories in L_P. For any two labels L_1 and L_2, we say $L_1 \sqsubseteq L_2$ iff L_1 contains all the integrity categories in L_2 and L_2 contains all the secrecy categories in L_1.

[1] The semantics of level **2** in a particular category in HiStar can be achieved by freely granting clearance in that category in DStar instead.

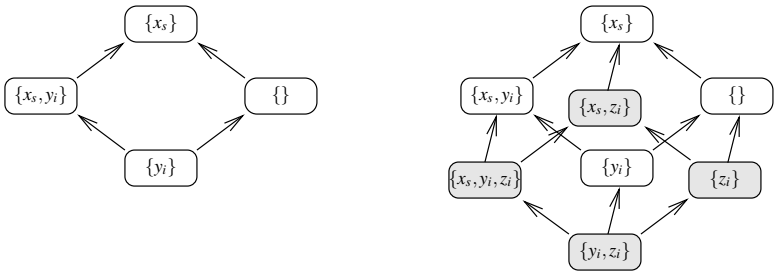

Figure 3.1: On the left, the lattice formed by one secrecy category, x_s, and one integrity category, y_i. On the right, another integrity category, z_i, is added, and the resulting lattice is shown, with new lattice points (labels) indicated by a shaded background. Arrows show pairs of labels where the "can flow to" $\sqsubseteq$ relation holds. Information can also flow transitively over multiple arrows at the same time.

With this notation, a sender S can send a message M as long as $L_S \sqsubseteq L_M$, and a receiver R can receive M as long as $L_M \sqsubseteq L_R$. As illustrated in Figure 3.1, labels form a lattice [9] under the $\sqsubseteq$ relation; arrows indicate the direction in which messages can be sent. Labels thus transitively enforce a form of mandatory access control.

Each process also has a second set of categories, O_P, called its ownership label. We say a process *owns* a category when its ownership label contains that category. Ownership confers the ability to remove any restrictions imposed by a category at the owner's discretion. Thus, we ignore the categories a process owns when determining what messages it can send and receive, and the rules become $L_S - O_S \sqsubseteq L_M - O_S$ and $L_M - O_R \sqsubseteq L_R - O_R$.

From the network's point of view, all access control is discretionary. Any host that receives data from the network can, at its discretion, retransmit the data with a different tracking label to an arbitrary destination. Therefore, all processes with direct network access must have the same tracking label, L_{net}, which in the absence of multiple networks is often empty, $L_{\text{net}} = \{\}$.

A process that sends and receives DStar messages over the network is known as an *exporter*. Exporters allow other processes that don't have direct network access nonetheless to send data and privilege across machines. Such processes typically have non-empty tracking labels; exporters arrange for the local operating system to enforce the restrictions implied by those labels. Specifically, an exporter's job is to ensure that, wherever data flows, the data's tracking label never drops secrecy categories or adds integrity categories except through the action of other processes owning those categories.

Consider the typical web server shown in Figure 3.2. The user data server sends one user's data to the application server in response to a query, but wants to ensure that the application code cannot misuse this data. To do this, it sends the user's data in a message M labeled $L_M = \{s\}$, where s is the user's secrecy

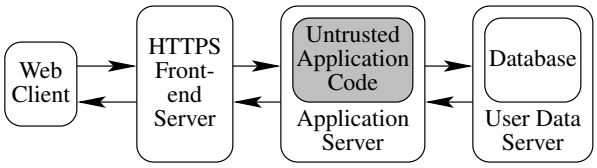

Figure 3.2: Typical architecture of a three-tiered web application. The three types of servers comprising the web application are the front-end server, the application server, and user data server. The shaded application code is typically the least trustworthy of all components, and should be treated as untrusted code to the extent possible.

category, to the exporter E_A running on the application server A. As long as no one has ever granted ownership of s to any process on A other than E_A, then E_A guarantees that M's contents will not affect any outgoing message not also labeled with s. (Conversely, if E_A transmits a message M' labeled $L_{M'} = \{i\}$ for some integrity category i not owned by any process other than E_A, this means the contents of M' cannot depend on any message A has received without i in its label.) To allow the data to eventually reach the user, the user data server grants ownership of s to the process handling the user's connection on the HTTP front-end server. This allows that process to send the application code's output over the network in an ordinary, non-DStar TCP connection.

DStar requires two things to enforce its guarantees. First, exporters must map DStar labels onto local operating system mechanisms that can enforce the appropriate restrictions. In the case of HiStar, this amounts to a mapping between DStar labels and HiStar labels, which we discuss in Section 3.2. Second, exporters need a way to ensure they are actually speaking to exporters on other machines and to determine which categories those exporters own. DStar uses cryptography for this purpose, as we will now describe.

3.1.1 Message Transfer Rules

Exporters that exchange messages are responsible for verifying each other's authority to participate in the exchange. Having a fixed network tracking label, L_{net}, simplifies the access checks. Before S sends a message M to R, it must ensure that R is trusted to safeguard M's privacy, which means checking $L_M - O_R \sqsubseteq L_R - O_R$. Since $L_S = L_R = L_{\text{net}}$, this becomes $L_M - O_R \sqsubseteq L_{\text{net}} - O_R$, or, when $L_{\text{net}} = \{\}$, simply $L_M - O_R \sqsubseteq \{\}$, a formal way of stating the requirement that R owns every secrecy category in M's tracking label. Similarly, before accepting the message, R must check that S can vouch for M's integrity by verifying that $L_{\text{net}} - O_S \sqsubseteq L_M - O_S$ or, if $L_{\text{net}} = \{\}$, just $\{\} \sqsubseteq L_M - O_S$ (meaning S owns all integrity categories in L_M). Thus, a key goal of the DStar protocol is to allow exporters to verify one another's ownership privileges.

$$
\begin{array}{rl}
\forall K & K \textit{ owns } K \\
\forall K, ID, T & K \textit{ owns } \langle K, ID, T \rangle \\
\forall K, K', x & (K \textit{ owns } K'), (K' \textit{ owns } x) \Rightarrow K \textit{ owns } x \\
\forall K, K', x & (K : K' \textit{ owns } x), (K \textit{ owns } x) \Rightarrow K' \textit{ owns } x
\end{array}
$$

Table 3.2: Rules for deriving the *owns* relation from DStar delegation certificates. K and K' are public keys, ID is a 64-bit identifier, and T is type of a category. x can be either a key or a category. $K : m$ denotes a certificate signed by K containing message m.

Each exporter has a public/private key pair, and exporters are named by their public keys. Exporters use their public keys for two main purposes: to authenticate communications with other exporters (by negotiating symmetric session keys), and to sign delegation certificates granting ownership to other exporters.

Category names in DStar are tuples of the form $\langle Creator, ID, Type \rangle$, where $Creator$ is the public key of the exporter that created this category, ID is an opaque 64-bit identifier, and $Type$ is either secrecy or integrity. An exporter by definition owns every category it creates, just as HiStar grants ownership of categories to the threads that create them. To create a category, an exporter simply chooses an ID value that it has not used before, which it can do by encrypting a counter with a block cipher.

Exporters can own both categories and other exporters. If exporter K owns K', then this means K owns every category owned by K'. An exporter can delegate ownership of any category or exporter it owns to any other exporter by signing a delegation certificate. A delegation certificate states that one exporter now owns either some other exporter or some category. Each certificate includes an expiration time to simplify revocation.

Table 3.2 summarizes the rules used to interpret these delegation certificates. These rules precisely define O_A, the set of categories owned by some exporter A.

When a sender checks that a message can flow to the recipient, the sender must not initiate any network communication to determine which categories the recipient owns. Otherwise, such communication could in itself convey the fact that a secret process is sending a message, thereby leaking information. Fortunately, embedding public keys in the names of categories allows the sender to verify that the receiver owns a category without any additional external information, based purely on self-authenticating delegation certificates. In the next section, we discuss how to distribute these certificates without introducing covert channels.

To this point, we have only defined the rules for when it is safe to send a message to an exporter. However, at a low level, network messages are sent to network addresses, not public keys. Any communication to look up the IP address of an exporter based on its key can in itself leak information. We therefore introduce *address delegation* certificates, which contain the exporter's current network address, signed with the exporter's key. An address delegation means that only the named exporter will learn of messages sent to the specified address.

Exporters currently distribute address delegations by periodically broadcasting them to the local-area network. Expiration times allow IP address reuse: after expiration, exporters will not connect to the old address, as doing so could leak information. In a complex network, broadcast would not be sufficient to distribute address delegations to all of the exporters. In that situation, we envision using an explicitly-trusted directory service, akin to DNS, that would map exporter keys to recent address delegation certificates for that exporter. We have not implemented this in our prototype.

In addition to transferring data, DStar provides privilege transfer. Each message M includes a set of categories O_M whose ownership should be granted to the message recipient. When sender exporter S sends message M to receiving exporter R, R verifies that S is authorized to grant the privileges specified by the message by checking that $O_M - O_S = \{\}$.

3.1.2 Sending Messages

DStar messages are sent to message *slots* on participating machines. A slot is similar to a port listening for requests from the network, but it also explicitly specifies the resources used to deliver the message, an important consideration to avoid covert channels. On HiStar, slots correspond to either gates or segments.

Consider a system which uses shared resources to queue incoming messages. A secret process could leak information by sending (or not sending) many messages to fill up some message queue; a non-secret process could send a message of its own, and learn information based on whether its message is processed or dropped. Explicitly naming resources allows the exporter to avoid such covert channels: as we will discuss later, in HiStar all resources are labeled, and the exporter only delivers messages to resources with a matching tracking label. Although the network itself is a shared resource, our goal is to make it the only shared resource; future work on explicit network resource allocation may help address that problem.

The encrypted message format used by exporters over the network is as follows:

```
struct wire_message {
    pubkey recipient_exporter;
```

```
    slot recipient_slot;
    label tracking;
    label ownership;
    label clearance;
    delegation_set dset;
    mapping_set mapset;
    opaque data;
};
```

The destination of the message is slot `recipient_slot` on `recipient_exporter`'s machine. The `tracking` label specifies information flow restrictions on the contents of the message. The recipient exporter will grant ownership of categories specified in the `ownership` label to the recipient slot when it delivers the message. The protocol also allows granting clearance privilege; it follows much the same rules as ownership. `dset` contains delegations proving to the recipient exporter that the sender owns all integrity categories in the `tracking` label, and all categories in the `ownership` and `clearance` labels. The `mapset` contains mappings between DStar categories and local security mechanisms on the recipient machine; we will discuss them in more detail in the next section. The payload is stored in `data`.

The exporter provides a single function to send messages:

```
void send(ip_address, tcp_port, wire_message, delegation_set, mapping_set);
```

Here, the `delegation_set` and `mapping_set` have the opposite roles from their counterparts in the `wire_message`. They must prove to the *local* exporter that it is safe to send the supplied `wire_message` to the recipient exporter. In particular, the `delegation_set` contains delegations that prove the recipient exporter owns all secrecy categories in the message tracking label, and the `mapping_set` allows the exporter to translate between the local security mechanism on the sending host and DStar categories. In addition, an address delegation, proving that the specified IP address and TCP port number speak to the recipient exporter, must be included in the `delegation_set`.

3.1.3 RPC

DStar provides a traditional RPC interface in an application library. This RPC library is not part of the trusted exporter, and instead runs in the application's address space and protection domain, using the exporter's message delivery interface for communication. Most importantly, the RPC library cannot violate any security guarantees enforced by the exporter.

When making an RPC call, the client library first creates a slot to receive the server's response, and then sends the request message to the server's slot. The request message specifies the response slot that

was just created by the client library, and delegation certificates and mappings for the server to send back a response. The server library sends its response to the slot specified in the request message, using delegation certificates and mappings provided by the client. In case of message loss, it is up to the client to either retransmit the request or return an error after a timeout. The server can implement a reply cache to provide at-most-once semantics.

Typically, the server's response message has the same tracking label as that of the client's request, and the response message implicitly acknowledges the successful delivery of the client's request message. However, the server may want to send a response with a different tracking label; for instance, it may add an additional secrecy category to the response message. This leads to the following complications.

First, the client must provide resources to accept a secret response message from the server. Second, the initial client thread that sent the request message will not be able to observe the more secret response message sent by the server. As a result, the server must send an explicit delivery acknowledgment to the client, with the tracking label of the original request, to prevent the client from retransmitting the request. The server must also eventually inform the initial client thread that it can reclaim the resources used to process the secret response; the server can batch these notifications over a long period of time to reduce covert channels. Finally, the server may be able to send multiple response messages to the client without the client's knowledge; this can be addressed with the help of the *guarded invocation* service described next. We will not discuss this style of RPC in more detail, because it is not used by any of the applications described in this dissertation.

3.1.4 Additional Services

DStar exporters provide some additional functionality, implemented as RPC servers on well-known slots.

The **delegation service** allows an owner of a category to create a delegation to another exporter, named by a public key; a signed delegation is returned to the client. If this service is compromised, an attacker can create arbitrary delegations for all categories owned by this exporter, compromising the security of any data that this exporter is trusted to handle. This service is fully trusted by the exporter.

The **mapping service** creates mappings between DStar categories and local operating system security mechanisms; it will be discussed in more detail in the next section. This service is also fully trusted.

The **container service** provides a way to allocate a new container in an existing one, with a new tracking label. This service is attested for, but not trusted by the exporter: its compromise will not violate the exporter's security guarantees.

The container service provides a timeout mechanism used to garbage-collect resources on the server in case of client failure during a series of RPC calls. A container can be created with a timeout, in which case a thread is started that will delete the container after the specified timeout. Ephemeral state, such as category mappings, associated with some higher-level transaction, is stored in this container. Even if the client fails, resources will be garbage-collected.

The **guarded invocation service** launches verified executables. A client can use this service to start a process with specified arguments and privileges on a remote machine, as long as a cryptographic checksum of the executable file on the remote machine matches the checksum provided by the client. The caller must provide resources to execute the resulting process. This service is used in bootstrapping, when only the public key of a trusted exporter is known. The exporter attests to the authenticity of this service, but does not trust the service itself.

3.2 HiStar Exporter

We first discuss the overall design of the exporter, inspired by the design of the HiStar kernel, and then describe how the exporter enforces information flow control on HiStar.

3.2.1 Overview

To reduce the effect of any compromise, the exporter *avoids superuser privilege by design*. We have already described the mutually-distrustful DStar network protocol, but the same principle applies on the local machine as well. The exporter runs as an ordinary process on HiStar without any inherent privileges from the kernel. The owner of each local category explicitly allows the exporter to translate between that category's restriction on the local machine and encrypted messages to other DStar exporters, by granting ownership of the local HiStar category to the exporter. The exporter uses this ownership to allow threads with non-empty tracking labels, which may not be able to send or receive network messages directly, to send and receive appropriately-labeled DStar messages.

Resource allocation can lead to unexpected covert channels, or worse yet, denial of service attacks, if not addressed by the fundamental design. The overall design of DStar, and that of the HiStar exporter, *labels all resources in the system*, and ensures that resource use and allocation are subject to the same information flow constraints as reading and writing.

Finally, to ensure that all data is appropriately labeled and that these labels are checked on access, *global data and resources are avoided*. Instead, the exporter is largely stateless, and requires that all state and resources necessary to send and receive messages be explicitly specified in each request. This aspect of the design is similar in spirit to the "no ambient authority" principle espoused by capability systems [17]. We will now describe how a stateless exporter can still maintain mappings between DStar and HiStar categories.

3.2.2 Category Mappings

Recall that one of the main tasks of the exporter is to translate between DStar categories and corresponding HiStar categories on the local machine. Since the exporter must be stateless, it is up to the users to supply these mappings in each message. However, these mappings are crucial to the security guarantees provided by the exporter—by corrupting these mappings, an attacker could convince the exporter to label an incoming secret message using a category owned by the attacker on the local machine, violating all security guarantees.

In the network protocol, exporters use signed certificates to get around this problem: users supply certificates to send each message, but exporters verify the signature on each certificate. On the local machine, exporters also need ownership of the local HiStar category in order to be able to manipulate data labeled with that category. Since the HiStar kernel only allows category ownership to be stored in thread or gate objects, the exporter fundamentally requires resources (for a kernel object) for each category it handles on the local machine.

Thus, for each mapping between a DStar category and a HiStar category, the exporter needs two things: a kernel object storing ownership of the local HiStar category, and some sort of a secure binding between the DStar and HiStar categories. The secure binding could be represented by a certificate, but since the exporter already needs a kernel object to store ownership, we store the secure binding along with that kernel object, and avoid the overhead of public key cryptography.

The objects representing a mapping between DStar category d and HiStar category c are shown in Figure 3.3. These objects are stored in a user-provided container, allowing users to manage their own resources, and making the exporter stateless. The exporter uses a pair of secrecy and integrity categories, e_s and e_i, to ensure the security of mappings, in the face of malicious users providing resources for these mappings. Gate g stores the exporter's ownership of category c. The gate has a *verify label* of $\{e_i\,\mathbf{0}, \mathbf{3}\}$ (not shown in the figure), ensuring that only the exporter, which owns e_i, can use this gate to obtain ownership

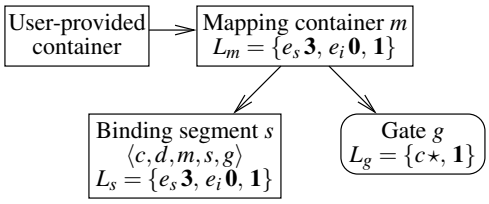

Figure 3.3: Objects comprising a mapping between DStar category d and local HiStar category c. Only the exporter owns secrecy and integrity categories e_s and e_i.

of c. A secure binding segment s stores the mapping tuple $\langle c, d, m, s, g\rangle$, consisting of the HiStar and DStar categories, and the object IDs of the objects that comprise the mapping.

Users must provide this mapping tuple, $\langle c, d, m, s, g\rangle$, in order to use a particular category mapping. The `mapping_set`, mentioned earlier in the `wire_message` and the `send()` function, is a set of such tuples. The exporter verifies the integrity of each mapping by checking that the supplied mapping tuple matches the contents of the binding segment, and that the label on the binding segment is correct. Since only the exporter owns e_i, no one else could have altered the binding segment, and the mapping is authentic.

The mapping service, briefly mentioned earlier, allows applications to create new mappings. This service can allocate a fresh HiStar category for an existing DStar category, or a fresh DStar category for an existing HiStar category. The exporter does not grant the freshly-allocated category to the caller; instead, if the caller owns the existing category, it can separately gain ownership of the new category through the mapping.

It is safe for anyone to create such fresh mappings: if they did not own the existing category, they will not own the newly-allocated category either, and will not be able to change the security policy set by the owner of the existing category. On the other hand, creating a mapping between an existing pair of HiStar and DStar categories requires the caller to prove ownership of both categories, by granting them to the mapping service. In all cases, the caller must provide the container to store the new mapping, and must grant the mapping service privileges to access this container.

The exporter does not check whether the sender can observe the parent container of the mapping container, a check typically done by HiStar to ensure the caller is authorized to observe the container's resources. This potentially opens up a covert channel when the mapping is deallocated from its parent container. However, we treat the mapping tuple as a capability, or pre-authorization, to observe whether the mapping has been deallocated. Thus, the covert channel can only be exploited by processes that can

already, in some way, communicate mapping tuples to one another. The HiStar kernel similarly treats containers as pre-authorizations to use resources that one would not be able to otherwise allocate. The pseudo-random 61-bit object IDs of the gate g and binding segment s prevent anyone from guessing a valid mapping tuple to learn when it gets deallocated.

3.2.3 Exporter Interface

To allow other processes to send DStar messages, the exporter on HiStar provides a well-known gate. A thread first writes the message it wants to send to a segment object, and then invokes the exporter's gate, specifying the object ID of the message segment and granting on gate invocation any privileges that it wants to send as part of the message. The exporter uses the *verify label* and *verify clearance* to check that the caller can read and write the supplied segment, and that the caller's tracking label is compatible with the tracking and ownership labels specified in the message, before sending the message.

When delivering remote messages to the local machine, the HiStar exporter supports two types of message slots: segment slots and gate slots. Only gate slots support privilege transfer, but they incur higher overhead than segment slots.

Segment slots name a segment by its 61-bit kernel object ID and the object ID of its parent container (required to name any object in HiStar). Before delivering a message to a segment slot, the exporter translates the DStar label of the message into a HiStar label, and ensures that the segment tracking label is the same as the message tracking label. This both provides access control and avoids resource covert channels. (The exporter also checks that the sender, represented by the message tracking label, can read the segment's parent container.) To deliver the message, the exporter writes the message into the segment, and uses a futex [13] to wake up any threads waiting for a message.

Gate slots name a gate kernel object and a container used for message delivery. To deliver a message to a gate slot, the exporter creates a segment containing the message in the slot's specified container, then creates a new thread in the same container, and uses this thread to invoke the slot's gate with all ownership privileges that were specified in the message. The tracking label of the segment and the thread are determined by translating the message's DStar tracking label into a HiStar label. As with segment slots, the exporter ensures that the caller, represented by the message tracking label, is allowed to modify and observe the container and observe the gate.

All broadcast address delegation certificates received by an exporter are made available to other threads on the same machine, by writing them to a well-known file with a tracking label of $\{e_i \mathbf{0}, \mathbf{1}\}$.

The file's tracking label ensures that only threads authorized to receive data from the network can read it, and only the exporter can write to it. This makes it easy to find address delegation certificates for local exporters without adding covert channels.

3.3 Implementation

The current implementation of the exporter comprises about 3,700 lines of C++ source code, and runs on both HiStar and Linux (though on Linux, all software must be trusted to obey information flow restrictions). The client library, trusted by individual processes to talk to the exporter on HiStar, is another 1,500 lines of C and C++ code. The exporter uses the libasync event-driven library [29] for processing network messages and cryptography, as well as libc and libstdc++, which dwarf it in terms of lines of code.

3.3.1 Privilege Management

The client RPC library uses unbound gates to store any privileges received from the server; such privileges are initially held by the incoming message thread. The initial client thread can then acquire these privileges by invoking the gate.

The exporter uses unbound gates to avoid the need for large labels representing all of its privileges. When a local thread invokes the exporter's gate to send a message, the thread may have secrecy categories in its tracking label, which prevent it from modifying the exporter's address space when it first enters. In this case, the thread tries to acquire ownership of all such categories by invoking the gates associated with each category mapping provided as part of the request. If it now owns all those categories, it can switch to the original exporter address space, and continue processing the message there; otherwise, it halts. The tracking label of any thread is therefore proportional to the number of categories in the tracking label of the message, rather than the total number of categories owned by the exporter.

3.4 Applications

To illustrate how DStar helps build secure distributed systems, we focus on two scenarios, based around web applications. We first show how the HiStar web server can be distributed over multiple machines like a typical three-tiered web application, providing performance scalability and strong security guarantees similar to those provided by a single HiStar machine. Second, we also show how, even in an existing web

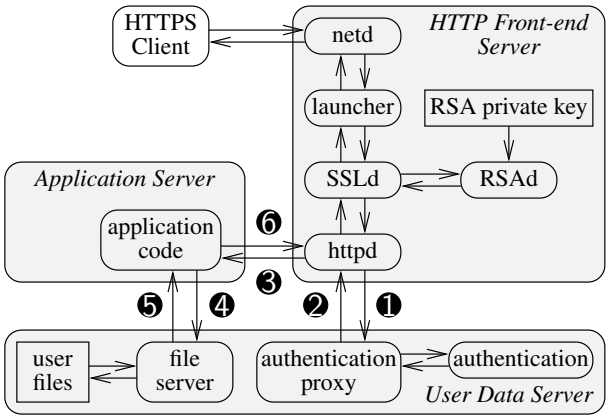

Figure 3.4: Structure of the same privilege-separated SSL web server running on multiple physical HiStar machines. Shaded boxes represent physical machines, and communication between these physical machines is done through the DStar exporters running on individual machines. Circled numbers indicate the order in which DStar messages are sent between machines. Not shown are DStar messages to create mappings on remote machines.

service environment, DStar can be used to improve security by incrementally adding information flow control for untrusted code.

Although our evaluation focuses on well-understood web applications, many distributed systems could benefit from specifying global security policies in terms of information flow. For example, a medical records system could use DStar as a mechanism to ensure patient privacy across a distributed system of servers and desktops. The policy controlling release of patient records would be implemented by a small trusted program with ownership of the patient's secrecy category.

3.4.1 Distributed Web Server

We have taken the HiStar web server, described earlier in Section 2.5.4, and used DStar to turn it into a typical three-tiered web application, as shown in Figure 3.4. The first tier, the HTTP front-end servers, run components responsible for accepting client connections and handling the HTTP protocol: the *launcher*, *SSLd*, *RSAd*, and *httpd*. The second tier, or application servers, run application-specific logic to handle user requests. Finally, the third tier, user data servers, store private user data and perform user authentication.

Servers in the first two tiers are largely stateless, making it easy to improve overall performance by adding more physical machines. This is an important consideration for complex web applications, where simple tasks such as generating a PDF document can easily consume 100 milliseconds of CPU time, when using GNU ghostscript for this purpose. The third tier, the user data servers, can also be partitioned over

multiple machines, by keeping a consistent mapping from each individual user to the particular user data server responsible for his data. Our prototype has a statically-configured mapping from users to user data servers, replicated on each HTTP front-end server.

These changes involved adding 740 lines of C++ code: a 140-line trusted authentication proxy, a 220-line untrusted RPC server to launch application code, a 100-line file server, trusted to preserve integrity but not secrecy, and finally 280 lines of code added to *httpd*.

Interactions between components on the same machine remain the same, with the same security properties as before. For instance, the RSA private key is kept private by the same *RSAd* process, providing only an RSA signing function. Interactions between components on different machines, on the other hand, now go through the exporters on the respective machines, but still maintain the same structure and security properties as on a single machine, with the exception that the exporters are now part of the trusted code base for the web server.

The one difference is authentication; we have not yet ported the HiStar authentication service to work over DStar, and instead rely on a trusted authentication proxy to invoke the HiStar authentication service locally on the user data server. *httpd* trusts the authentication proxy to keep the user's password secure, but guarded invocation can ensure that the user's password is only passed to an approved authentication proxy binary. The next subsection discusses how the web server uses DStar to provide similar security properties in a distributed system as we achieved on a single machine, by trusting the exporter code on each machine.

3.4.2 Using DStar in the Distributed Web Server

To use DStar, applications, such as our distributed web server, must explicitly manage two important aspects of the distributed system. First, applications must explicitly define trust between the different machines in the distributed system, by creating the appropriate delegation certificates. Second, applications need to explicitly allocate resources on different machines—including containers and category mappings—to be able to communicate between machines and execute code remotely. The rest of this section describes how the distributed web server addresses these issues.

The key trust relation in our distributed web server concerns the individual users' secrecy and integrity categories, or in other words, what machines are authorized to act on behalf of what users. All user categories in our design are initially created, and therefore owned, by the exporter on that user's data server. When the authentication proxy receives the correct user password in Step 1, it asks the local

exporter to create a short-lived delegation certificate, valid only for a few minutes, for the user's secrecy and integrity categories to the exporter on httpd's front-end machine. Short-lived certificates ensures that, even if other machines are compromised, they can only subvert the security of users that are currently using, or have recently used those machines. The authentication proxy sends these certificates, along with granting ownership of the user's categories using the `ownership` field of the message, to httpd in Step 2.

Although httpd receives ownership of the user's categories, it does not act on behalf of the user directly. Instead, httpd passes ownership of the user's categories to the application server in Step 3, where application code uses them in Step 4 to communicate with the user data server. httpd asks the exporter on its front-end machine to generate these delegation certificates to the application server. To be considered valid, these certificates must be presented together with a chain of certificates up to the category's creator—the user data server—proving that the front-end machine is authorized to delegate ownership in the first place. Since this chain includes the initial, short-lived certificate from the authentication proxy, malicious exporters cannot extend the amount of time they can act on the user's behalf.

The distributed web server must also explicitly provide resources for all messages and processes. Since the launcher process drives the execution of each user request, it starts out with ownership of a global resource category, r_i^G, which gives it access to resources on all other machines (we will discuss this category in more detail in the next subsection). Each application and user data server has a container labeled $\{r_i^G\}$, known to the launcher process. The container's label allows the launcher process to use that container's resources.

When the launcher starts httpd, it grants it ownership of r_i^G, and gives it the names of containers on all other servers. When httpd communicates with the authentication proxy in Step 1, for example, it uses one of these containers, along with its ownership of r_i^G, to send the request message.

A notable case arises in Step 4, when the application code wants to communicate with the file server. Although httpd could grant the application code ownership of r_i^G, the application code would not be able to use the resources of a container labeled $\{r_i^G\}$ on the user data server, because the application code is itself labeled $\{u_s\}$, and allowing it to write to that container would constitute a covert channel. Instead, httpd pre-allocates a sub-container with a label of $\{u_s, u_i\}$ on the user data server, using that machine's resource allocation service, and passes the name of this container to the application code in Step 3. The application code can use this container to communicate with the file server, but cannot leak the user's private data through resource covert channels.

Although in our prototype the application code communicates with only one user data server, a more complex application can make use of multiple data servers to handle a single request. Doing so would

require the application code to present delegation certificates and have access to resources on all the data servers it wants to contact. To do this, httpd could either pre-allocate all such delegations and resources ahead of time, or provide a callback interface to allocate resources and delegations on demand. In the latter case, *httpd* could rate-limit or batch requests to reduce covert channels.

3.4.3 Bootstrapping and Replication

Though the HTTP front-end servers and the application servers are largely stateless, a bootstrapping mechanism is still needed to add a new server to an existing distributed system. In the case of HTTP front-end servers, the SSL private key must also be securely distributed to the new server to start a new *RSAd* process; we want to ensure that the private key is only revealed to an *RSAd* process, and not to any other process that may be running on the remote machine.

For analogy, consider the process of adding a new machine to an existing Linux cluster. An administrator might typically install Linux, then from the console set a root password, configure an ssh server, and (if diligent about security) record the ssh host key to enter on other machines. From this point on, the administrator can access the new machine remotely, and might use a tool such as *rdist* or *rsync* to copy over configuration files and application binaries. The ability to copy private data to the new machine safely stems from knowing its ssh host key, while the authority to access the machine in the first place stems from knowing its root password.

To add a new physical machine to a DStar cluster requires similar guarantees. Instead of an ssh host key, the administrator just records the exporter's public key, but the function is the same, namely for other machines to know they are talking to the new machine and not an impostor. However, DStar has no equivalent of the root password, and instead uses categories.

In fact, the root password serves two somewhat distinct purposes in Linux: it authorizes clients to allocate resources such as processes and memory on the new machine—i.e., to run programs—and it authorizes the programs that are run to access and modify data on the machine. DStar splits these privileges amongst different categories.

When first setting up the DStar cluster, the administrator creates an integrity category r_i on the first HiStar machine, and a corresponding DStar category r_i^G (G for global). r_i^G represents the ability to allocate and deallocate all resources used by the distributed web server on any machine. It can be thought of as the "resource allocation root" for this particular application. However, there is no equivalent "data access root." Instead, different pieces of data are protected by different categories.

When configuring the new machine, the administrator must allow r_i^G to allocate resources. There is a bootstrap procedure in which the administrator gives the name of category r_i^G to the exporter, the exporter allocates a local HiStar category r_i', creates a new container protected by r_i', and stores a mapping between r_i^G and r_i' in that container. Finally, the exporter announces the mapping it just created for r_i' and the object ID of the new container; the mapping is self-authenticating, and need not be securely transferred.

Processes on other machines can now copy files to the new machine and execute code there if they own r_i^G and know the mapping from r_i^G to r_i', much as the ssh client can on Linux if it knows the new machine's root password. The difference is that a process that owns r_i^G and other categories can use the other categories to create files that cannot be read and written just by virtue of owning r_i^G.

For example, to replicate *RSAd*, the administrator creates a sub-container in the machine's initial container, protected by a fresh integrity category, creates appropriate mappings and delegations for this category, and passes ownership of this category to an *RSAd* replication daemon running on the local machine, along with the public key of the new server where the new *RSAd* should run. The replication daemon uses the provided container to create mappings and delegations for the private key's secrecy category, and uses guarded invocation to invoke an identical *RSAd* binary on the remote host.

The private key is sent to the new machine protected by its DStar secrecy category, rsa_s^G. Ownership of this secrecy category is granted to the new machine's exporter and *RSAd* process, but the latter is conditional on guarded invocation (in case an attacker has tampered with the *RSAd* binary on the new machine). Note that the original process that granted the *RSAd* replication daemon access to a container on the new machine cannot read the private key now stored there, because it does not own rsa_s^G. Both the replication daemon and the guarded invocation service, consisting of 120 and 200 lines of C++ code respectively, are trusted to keep the private key secure, in addition to *RSAd* itself.

Once *RSAd* is running on the new machine, the administrator starts the *launcher*, also with guarded invocation, and grants it ownership of categories it can use to communicate with the user data and application servers. The *launcher*, in turn, starts *SSLd* and *httpd* as incoming client connections arrive, and grants *httpd* the same categories so it can interact with user data and application servers directly. HiStar's system-wide persistence eliminates the need for a trusted process to start *RSAd* and *launcher* when the machine reboots.

The current bootstrap procedure is tedious, requiring the manual transfer of category names and public keys. In the future, we envisage a setup utility that uses a password protocol like SRP [57] to achieve

mutual authentication with an installation daemon and automate the process. Alternatively, hardware attestation, such as TCPA, could be used to vouch for the fact that a given machine is running HiStar and a DStar exporter with a particular public key.

3.4.4 Heterogeneous Systems

To illustrate how DStar can facilitate incremental deployment, we show how Linux can use HiStar to execute untrusted perl code with strong security guarantees. To this end, we have implemented a DStar RPC server running on HiStar, which takes as input the source code of a perl script and the input data for the perl script, and returns two outputs: the return status of the perl process, and its output, when run using the provided script and input data. The DStar exporter translates any information flow restrictions and privileges supplied by the caller into HiStar labels, which the HiStar kernel then enforces.

Using the Unix port of DStar, a Linux machine can use this perl service to execute untrusted perl code, with well-defined security properties beyond those of a typical sandbox. It is up to the Linux machine to specify how different instances of perl can share data, by specifying the security policy using secrecy and integrity categories in the tracking label of the request message. For instance, to ensure each request is processed in complete isolation, a new secrecy category is used for each request.

If different scripts running on behalf of the same user should be allowed to share data, such as by storing data in the file system on the HiStar machine, the same secrecy category should be used for each request made on behalf of a given user. Most sandboxing mechanisms on Linux offer limited facilities for secure, persistent storage.

One distinction in using DStar from a Linux machine rather than HiStar is that the caller must reason about labels in terms of DStar categories, rather than local HiStar categories; the Linux kernel provides no local category abstraction of its own.

A Linux machine can also be used as a database for HiStar machines, as long as all software running on Linux is fully trusted to keep information flow restrictions, for example by storing the tracking label of each database row in a special column. While such an approach may open certain covert channels inherent in Linux's implicit resource allocation, it may be an appealing alternative for incremental deployment.

Although not evaluated in this dissertation, we believe DStar could facilitate safe communication between currently-disparate systems that provide strong isolation, including label-based operating systems like HiStar, capability-based operating systems like EROS, and programming languages like Jif.

Work-load	Front end servers Application servers User data servers	1 1 1	2 1 1	3 1 1	1 2 1	1 3 1	1 4 1	1 1 2	2 1 2	3 1 2	1 2 2	1 3 2	2 3 1	2 3 2	1	No DStar	No PS	Linux
PDF	Max throughput (req/sec)	4.7	4.7	4.6	8.9	10.8	11.7	4.8	4.9	4.8	9.3	11.7	11.2	13.9	3.4	4.4	4.8	6.0
	Min latency (msec)	301													293	226	207	173
cat	Max throughput (req/sec)	11.7	15.8	16.3	11.8	11.8	11.8	14.3	24.6	27.8	14.7	14.7	15.7	25.7	9	25.3	42.4	93.6
	Min latency (msec)	110													110	42	25	17

Table 3.3: Maximum throughput and minimum latency achieved by different web servers and configurations under two workloads. The PDF workload generated a 2 page PDF document, and the cat workload ran `cat` to generate an 8KB text document. The column labeled Linux reflects Apache. "No PS" ran our web server on HiStar in a single address space, without any privilege separation. "No DStar" ran the privilege-separated web server from Section 2.5.4. The "1" column ran the distributed web server all on one machine. Other columns represent different numbers of physical machines used for the distributed web server.

3.5 Performance

To quantify the overheads imposed by DStar, we evaluate the performance of our web server and perl service under various conditions. The overheads introduced by both HiStar and DStar in the web server are acceptable for compute-intensive web applications such as PDF generation, where the performance is close to that of a Linux system. On the other hand, our web server delivers static content much more slowly than Linux.

Benchmarks were run on 2.4GHz AMD Athlon64 machines with 1MB of CPU cache, 1GB of main memory, and a switched 100Mbps Ethernet. In benchmarks requiring multiple machines, some of the machines had a similar processor with 512KB of CPU cache. For web server comparison experiments, we used Apache 2.0.55 on Ubuntu 6.06 with kernel version 2.6.15-26-amd64-generic. The PDF workload used a2ps version 4.13b and ghostscript version 8.54. Xen experiments used Xen version 3.0.4 and kernel 2.6.16.33-xen for all domains. Web servers used OpenSSL 0.9.8a with 1024-bit certificate keys; DStar exporters used 1280-bit Rabin keys.

3.5.1 Application Performance

To evaluate the performance of applications running on our web server, we approximated a realistic service using a PDF generation workload. For each request, an HTTPS client connected to the server and supplied its user name and password. The web server generated a 2-page PDF document based on an 8KB user text file stored on the server, and sent it back to the client. Clients alternated between two different users; since the web server did not cache any authentication state, this did not skew performance. With multiple front-end servers, clients used a round-robin selection policy. We measured the minimum

latency observed when a single client was active, and the maximum throughput achieved with the optimal number of clients. The optimal number of clients varied for different configurations, but was generally proportional to the number of servers.

Table 3.3 shows the results of this experiment. Linux provides 25% more throughput than a non-privilege-separated web server running on HiStar; the difference is largely due to the overhead of fork and exec on HiStar. Privilege separation of the web server on HiStar adds a further 8% penalty. Running the distributed web server on a single machine shows the CPU overhead imposed by DStar, although such a configuration would not be used in practice.

Additional front-end and user data servers did not improve the performance of this workload when only one application server was in use, as the workload is largely CPU-bound on the application server. Adding a second application server improves throughput by 90%, and a third application server brings 40% more throughput. At this point the bottleneck has shifted over to the front-end servers. Using two front-end and user data servers avoids bottlenecks there, and in that case, three application servers deliver 2.95 times the throughput of a single application server.

3.5.2 Web Server Overhead

To more closely examine the overheads imposed by our web server, we replaced the CPU-intensive PDF workload with a `cat` process which simply reads an 8KB user file from the server; the results are shown in Table 3.3. Both the privilege-separated and non-privilege-separated web servers on HiStar have much lower performance than Apache. Although Apache can achieve even greater throughput serving static content without using `cat`, we wanted to measure the overhead of executing application code. The lower performance reflects that the design of our web server is geared towards isolation of complex application code; running simple application code incurs prohibitively high overhead.

Although the distributed web server has relatively low performance on a single machine, it achieves good performance scalability when adding more physical machines. Unlike in the PDF workload, the bottlenecks here are the front-end and user data servers. Doubling the number of front-end and user data servers more than doubles the overall throughput; we believe this anomaly is caused by batching of TCP/IP and DStar messages on the application server, which reduces context switching overhead. Using two front-end or two user data servers alone does not achieve the same performance improvement, suggesting that both are bottlenecks.

	lwIP on HiStar	lwIP on Linux	Linux native
Throughput	267 req/sec	238 req/sec	5595+ req/sec
Request latency	3.8 msec	4.7 msec	1.4 msec

Table 3.4: Throughput and latency of an echo server; each request opens a new connection and sequentially sends and receives five 150-byte messages. When the server used the Linux TCP/IP stack, the client saturated before the server. lwIP runs faster on HiStar due to direct access to the network device.

We believe that our web server's low performance is in part due to the high latency and low throughput of lwIP and its sockets API, since it was optimized to conserve memory on embedded devices. We ported lwIP to run in user space on Linux, and measured the throughput and latency of requests to a TCP echo server running with the native Linux TCP/IP stack, lwIP on Linux, and lwIP on HiStar. Table 3.4 shows the results: lwIP achieves similar performance on both Linux and HiStar, while the native Linux TCP/IP stack significantly outperforms lwIP. Running in user space is unlikely to be key to lwIP's low performance; Click [22] showed that user-level TCP implementations can perform competitively. We hope to achieve better performance in the future by running the Linux TCP stack in the *netd* process on HiStar.

3.5.3 Privilege-Separation on Linux

Is it possible to construct a simpler, faster privilege-separated web server on Linux that offers similar security properties? We constructed a prototype, running separate *launcher*, *SSLd*, *RSAd*, and Apache processes, using chroot and setuid to isolate different components from each other. This configuration performed similarly to a monolithic Apache server. However, to isolate different users' application code from one another, Apache needs access to setuid, and needs to run as root, a step back in security. We can fix this by running Apache and application code in a Xen VM; this reduces the throughput of the PDF workload to 5.5 req/sec. Even this configuration cannot guarantee that malicious application code cannot disclose user data; doing so would require one VM per request, a fairly expensive proposition. This suggests that the complexity and overhead of HiStar's web server may be reasonable for the security it provides, especially in a distributed setting.

3.5.4 Linux Integration

Table 3.5 shows the latency and throughput of running a simple perl script that prints "Hello world" on Linux, on HiStar, and finally on HiStar invoked using DStar from a Linux machine. A fresh secrecy category was used for each request in the latter case. This simple perl script provides a worst-case scenario,

	Linux	HiStar	HiStar using DStar
Throughput (req/sec)	349	161	52
Latency (msec)	2.9	6.5	96

Table 3.5: Throughput and latency of executing a "Hello world" perl script locally on Linux and HiStar, and remotely on a HiStar machine invoked from Linux using DStar.

by incurring all of the overhead of invoking perl with little actual computation to amortize over; perl scripts that perform actual work would fare much better. The lower performance of perl on HiStar is largely due to the overhead of emulating fork and exec system calls. DStar incurs a number of round-trips to allocate a secrecy category and create a container for ephemeral call state, which, combined with lwIP's high latency, contributes to a significantly higher request latency. Comparing the throughput of perl scripts on HiStar running locally and remotely shows that DStar adds an overhead of 13 msec of CPU time per request, which may be an acceptable price for executing arbitrary perl code with well-defined security properties from a Linux machine.

Chapter 4

Discussion

4.1 Data Control Idioms

HiStar and DStar applications that have been developed so far tend to use information flow control mechanisms in a few common patterns. This section shows these patterns and how they are used by the applications described earlier. Most of these patterns can be combined to achieve multiple security properties at the same time, by simply combining the label components from the individual patterns.

4.1.1 Discretionary Access Control

The Unix library on HiStar extensively implements discretionary access control using labels, by allocating a pair of categories, r and w, for each protection domain. For example, each user, each process, and each file descriptor has its own pair of discretionary protection handles. To prevent other threads from reading or writing protected objects, the object is labeled $\{r\mathbf{3}, w\mathbf{0}, \mathbf{1}\}$. Read-only protection, allowing anyone to read but not write the object, is achieved by using a tracking label of $\{w\mathbf{0}, \mathbf{1}\}$. Discretionary access is conveyed by granting either $\{r\star\}$ to give read access or $\{r\star, w\star\}$ to give read-write access. For example, to share a file descriptor between multiple processes or threads, ownership of that file descriptor's handles is granted to all of the threads using the file descriptor.

4.1.2 Secret Execution

Secret execution allows executing untrusted code on potentially private data while ensuring that the data and privileges given to this untrusted code are not leaked. In other words, any information about the execution of this untrusted code, including whether it was executed at all, is secret. This pattern is used to isolate untrusted code in both the virus scanner example presented in Chapter 1 and the *SSLd* daemon used by the web server. A fresh category, s, is allocated to isolate the untrusted code, and the code is executed with a tracking label of $\{s\mathbf{3}, \mathbf{1}\}$. Only threads labeled $\{s\star\}$, initially just the thread that allocated category s, can receive any information from the isolated, untrusted code, and as a result, they have full control over what data or privileges the untrusted code can expose.

4.1.3 Export Protection

Export protection provides an intermediate level of protection between allowing or disallowing read access to private files. If a file is read-protected with a tracking label of $\{r\mathbf{3}, \mathbf{1}\}$, untrusted code can be given limited access to this file by granting a clearance of $\{r\mathbf{3}, \mathbf{2}\}$, instead of a tracking label of $\{r\star, \mathbf{1}\}$, which would have given full discretionary access. This allows the untrusted code to read the file contents, but does not allow it communicate the contents to anyone outside of the system (except through threads labeled $\{r\star, \mathbf{1}\}$). This pattern is used in the HiStar web server to execute untrusted web application code with access to the user's private files, while ensuring that the application code cannot disclose the contents of the user's files to anyone outside of the system.

4.1.4 Taint Tracking

Similar to export protection, taint tracking prevents potentially sensitive data from leaving the system except through a small amount of privileged code. The main difference from export protection is in how this mechanism is used by the applications. Unlike export protection, taint tracking allows almost any thread to raise its tracking label to become tainted (using level **2** in HiStar), and the privileged code only ensures that the tainted data leaves the system in an appropriate way. For example, the VPN application uses two categories, i and v, to label data from the Internet and from the VPN respectively: data from the Internet is labeled $\{i\mathbf{2}, \mathbf{1}\}$ and data from the VPN is labeled $\{v\mathbf{2}, \mathbf{1}\}$. Internet data cannot not be sent out over the VPN, and vice-versa, without passing some additional sanity checks, such as a virus scanner.

4.1.5 Mutual Secrets

A variation of secret execution, this pattern allows the executed code to keep its own secret information that is not accessible to the caller. For example, Section 2.4.5 described a timestamped signature daemon that keeps a private signature key, but allows other processes to invoke it and generate signatures on their own data. Other processes cannot access the secret signature key directly, and the signature daemon code cannot launder the secret data supplied to it by other processes, since it is invoked with a tainted tracking label. The executed code is trusted to maintain the secrecy of its own private information, such as the private key in the signature daemon example. This pattern is used by the login process to verify the user-supplied password, and by the *RSAd* daemon in the web server, which holds the SSL certificate private key in secret.

A daemon process can allocate a category d to protect the secrecy of its own data, and store its data in a segment labeled $\{d\,\mathbf{3},\,\mathbf{1}\}$. The daemon then creates a gate with label $\{d\star,\,\mathbf{1}\}$ to allow other processes to invoke it on their private data. The caller would then allocate a fresh category s to protect the secrecy of its data, and invoke the daemon's gate on its private input data with a label of $\{d\star,\,s\,\mathbf{3},\,\mathbf{1}\}$. This allows the daemon to read both its data and the caller's data, but only return its results back to the caller. The daemon cannot export a copy of the caller's private data or store a copy in its own container.

4.1.6 Combining Privilege

Privilege combination allows two parties to combine their privileges to perform a certain operation, when neither trusts the other one with its privileges outright. For example, this pattern is used by the login process to create the retry counter segment, using the privileges of both the login process and the user authentication daemon. To combine privileges, both parties agree ahead of time on a short piece of code that they want to execute. The first of the two parties creates a read-only code segment containing this code, along with a gate that invokes this code segment with the first party's privileges. The second party verifies that the gate, address space, and segment contain the agreed-upon code and are marked read-only, and invokes the gate with its own privileges. The outcome is the execution of the read-only code segment with the privileges of both parties. Either party can potentially abort the execution, but cannot affect it in any other way.

4.1.7 Inward Confinement

Inward confinement can be used to ensure that executing code does not receive any unwanted external inputs, either from an attacker or some other colluding process. This can be useful in implementing a stronger form of a *chroot* jail, ensuring that an application runs in isolation, perhaps in combination with the secret execution pattern described earlier. A category i is allocated to provide inward confinement, and the confined code is executed with a tracking label of $\{i\mathbf{0}, \mathbf{1}\}$ and a clearance of $\{i\mathbf{0}, \mathbf{2}\}$. All approved inputs to the executing code are labeled $\{i\mathbf{0}, \mathbf{1}\}$. The tracking labels of most other objects in the system have a higher level in category i (typically level $\mathbf{1}$), precluding the executed code from being influenced by those objects.

4.2 Limitations

We believe HiStar and DStar provide a good environment to develop secure applications with small trusted code size. Nonetheless, the systems have limitations both in terms of functionality and security. Some of these limitations are artifacts of the implementation that we hope to correct, while others are more fundamental to the approach.

Users familiar with Unix will find that, though HiStar resembles Unix, it also lacks several useful features and changes the semantics of some operations. For example, HiStar does not currently keep file access times; although possible to implement for some cases, correctly tracking time of last access is in many situations fundamentally at odds with information flow control.

Another difference is that *chmod*, *chown*, and *chgrp* revoke all open file descriptors and copy the file or directory. Because each file has one read and one write category, group permissions require a file's owner to be in the group. There is no file execute permission without read permission, and no setuid bit (though gates arguably provide a better alternative to both). Several other facilities are missing, though we hope to add them, including support for system-wide backup and restore, and a user-level trampoline mechanism to allow upgrading of software behind gates (since gate entries are fixed).

Though HiStar is intended to allow administration without a superuser, we do not yet have experience administering a production HiStar system. However, we believe that to the extent it is needed, superuser privilege should be implemented by *convention*—explicitly granting most privilege to the root user—not by *design*. A HiStar administrator can still revoke all resources by virtue of having write permission on the

root container. This provides a worst-case answer to uncooperative users that refuse to grant the necessary privilege to root.

While the HiStar kernel automatically provides consistency across kernel crashes and restarts, a crashed or killed process can leave locked mutexes, such as the directory segment mutex. We currently do not recover from such problems, but foresee two potential solutions. The first is to do write-ahead logging in memory; given some way of detecting a dead or crashed process—for example, through timeouts—other processes can recover the directory segment. The second is to prevent the thread from being killed while it is holding the directory mutex, by adding a hard-link to it in the directory container. If the thread is unreferenced from other containers, it will continue executing until removing itself from the directory container.

Because Asbestos labels are more general than capabilities, they allow multiple objects to be protected by the same category and multiple categories to place restrictions on the same object. Users familiar with capability systems will rightfully object that protecting multiple objects with the same category limits the granularity at which privileges can be enumerated. HiStar can be used like a capability system by allocating a new category pair for every object, but our Unix library does not do this. However, as the VPN example showed, HiStar has the advantage of allowing new policies to be overlaid on existing software, which cannot be done as easily in pure capability systems.

One security limitation is that HiStar does not support CPU quotas, though we hope to add these using the container hierarchy. A more serious problem we do not know how to solve is covert timing channels. For example, in simple tests, we can leak 20 bits/second reliably though a wall-banging attack on HiStar, and suspect a more clever attacker would improve on our number by an order of magnitude. Moreover, many network services have to offer low response latency, and as a result, it becomes increasingly practical to leak information to outside observers by modulating response time. However, covert channel mitigation is outside the scope of this work.

DStar is particularly susceptible to covert timing channels that arise because of shared network resources. Additionally, an active attacker can do things like spoof ARP replies and overflow MAC address tables or even flood the network to glean information about the communication patterns of exporters. Using ordinary Ethernet switches, DStar must partially trust the network. This is somewhat reasonable if all malicious code is contained on HiStar machines, but dangerous should malicious code in HiStar collude with a Linux box on the same network to leak information. In future work, we intend to integrate DStar with network switches that can better conceal communication patterns [6].

The current prototype of DStar can only enforce information flow restrictions using HiStar. However, capability-based operating systems, such as KeyKOS [4] and EROS [47], can also be used to provide strict program isolation, albeit with an *a priori* partitioning of capabilities, which restricts program structure. A DStar exporter could run on EROS and enforce information flow constraints by partitioning capabilities according to labels. Once a process starts running with some tracking label, it can never change its tracking label, since there is no mechanism to revoke capabilities granting write access to files with a tracking label lower in the lattice than the new tracking label. A DStar exporter on a capability-based system would not be able to grant clearance (the privilege to raise one's tracking label). Wrapping each capability in a proxy that checks tracking labels would effectively result in a label-based system.

Chapter 5

Related work

There has been significant prior work on information flow control and controlling execution of untrusted code. This chapter discusses the most relevant related work in operating systems, networks, distributed systems, and programming languages.

5.1 Operating Systems

HiStar was directly inspired by Asbestos [10], but differs in providing system-wide persistence, explicit resource allocation, and a lower-level kernel interface that closes known covert storage channels and makes all information flow explicit. Unlike Asbestos, which is a message-passing system, HiStar relies heavily on shared memory. The HiStar kernel provides gates, not IPC, with the important distinction that upon crossing a gate, a thread's resources initially come from its previous domain. By contrast, Asbestos changes a process's tracking label to track information flow when it receives IPCs, which is detectable by third parties and can leak information. Asbestos highly optimizes comparisons between enormous labels, which so far we have not done in HiStar.

HiStar controls information flow with mandatory access control (MAC), a well-studied technique dating back decades [3]. The ADEPT-50 dynamically adjusted labels (essentially taint tracking) using the High-Water-Mark security model back in the late 1960s [24]; the idea has often resurfaced, for instance in IX [32] and LOMAC [14]. HiStar and its predecessor Asbestos are novel in that they make operations such as category allocation and declassification available to application programmers, where previous OSes reserved this functionality for security administrators. Decentralized declassification allows novel

uses of categories that we believe promote better application structure and support applications, such as web services, not targeted by previous MAC systems.

Superficially, HiStar resembles the capability-based KeyKOS [4] system and its successor EROS [47]. Both systems use a small number of kernel object types and a single-level store. HiStar's container abstraction is reminiscent of hierarchical space banks in KeyKOS. However, while KeyKOS uses kernel-level capabilities to enforce labels at user-level, HiStar bases all protection on kernel-level labels. The difference is significant because labels specify security properties while imposing less structure on applications—for example, an untrusted thread can dynamically alter its tracking label to observe secret data, which has no analogue in a capability system.

HiStar has no superuser. A number of previous systems have limited, partitioned [32], or virtualized [40] superuser privileges. Several operating systems including Linux support POSIX capabilities, which can permit some superuser privileges while disabling others.

Plan 9 [39] also has no superuser. Administrative tasks such as adding users can only be performed on the file server console, virtually eliminating the threat of network break-ins. On workstations, however, the console user has special privileges, and on compute servers a pseudo-user named "bootes" does. Plan 9 provides a complete, working system with a trusted computing base many times smaller than comparable operating systems. It also provides per-process file namespaces, which inspired HiStar's user-level mount table segments. However, Plan 9 was never intended to support MAC.

HiStar uses gates for protected control transfer, an idea dating back to Multics [45]. However, HiStar's protection domains are not hierarchical like Multics rings. HiStar gates are more like doors in Spring [16].

Singularity [18] provides programming-language-based security without an underlying operating system. Somewhat like containers, Singularity addresses coherent resource deallocation with a new abstraction called Software-Isolated Processes (SIPs). Singularity does not provide MAC, however.

SELinux [26] lets Linux support MAC; like most MAC systems, policy is centrally specified by the administrator. In contrast, HiStar lets applications craft policies around their own categories of information. Retrofitting MAC to a large existing kernel such as Linux is potentially error-prone, particularly given the sometimes ill-specified semantics of Linux system calls. HiStar's disciplined, small kernel can potentially achieve much higher assurance at the cost of compatibility.

Extensions to Linux such as [49] make it easier for applications to use the operating system's protection mechanisms, namely user IDs, to create their own protection domains. Unlike HiStar, these protection domains only provide discretionary access control, and cannot prevent the disclosure of sensitive information by malicious code.

5.2 Secure Networks

Multi-level secure networks [2, 12, 31, 48] enforce information flow control in a trusted network, but provide very coarse-grained trust partitioning. By comparison, DStar functions even in an untrusted network such as the Internet, at the cost of introducing some inherent covert channels, and allows fine-grained trust to be explicitly configured between hosts. Using a secure, trusted network would reduce covert channels introduced by DStar. There has also been significant work on ways to minimize the rate of such channels, for instance by introducing noise or randomness [19].

Unlike multi-level secure networks, DStar does not support labeling a machine without giving it ownership privilege: in other words, an exporter's tracking label is always the empty set. Providing a non-empty tracking label would require a trusted component to act as a proxy for the machine, ensuring that any packets sent or received by the machine are consistent with its current tracking label. This can be done either with support from the network, or by explicitly forwarding messages through a proxy trusted to maintain the tracking labels of machines it is proxying.

5.3 Distributed Systems

Shamon [30] is a distributed mandatory access control system that uses a shared reference monitor to enforce information flow policies between virtual machines. In contrast, DStar avoids the centralized authority of a shared reference monitor, making it practical to build distributed MAC systems that span administrative domains. Combining DStar with HiStar allows for policies to be applied to fine-grained objects such as files or threads, as opposed to entire VMs.

A number of systems, including Taos [56], have mechanisms for access control in a distributed system. However, none of them can enforce information flow control. Unlike capability-based operating systems, distributed capability systems such as Amoeba [52] cannot isolate untrusted code; a malicious program can always manufacture capabilities to contact a colluding server. The Taos *speaks-for* relation inspired the much simpler DStar *owns* relation, used to define discretionary privileges between exporters.

Jaeger et al [21] used IPsec to label network communications between SELinux machines, but did not provide an overall security policy for the distributed system, and relied on external mechanisms for establishing trust and bootstrapping. Labeling IPsec keys is similar to how the DStar exporter translates between local labels and authenticated encryption of network messages; instead of associating labels with encryption keys, DStar transfers the label in each message. Security mechanisms provided by SELinux

use a global per-machine policy; DStar allows anyone, including applications, to define new security policies at runtime.

5.4 Programming Languages

Decentralized declassification, while new in operating systems, was previously provided by programming languages, notably Jif [7, 35]. There are significant differences between a language and an operating system. Jif can track information flow at the level of individual variables and perform most label checks at compile time. It also has the luxury of relying on the underlying operating system for bootstrapping, storage, trusted input files, administration, etc., which avoids many issues HiStar needs to address.

Jif labels allow different principals to express their security concerns by specifying what other principals are allowed to read or write certain data. A DStar category roughly corresponds to a pair of Jif principals: an owner principal p_o and a tracking principal p_t. For each piece of data, each owner principal p_o can add either no restrictions, or require anyone reading the data (or writing, depending on the type of the category) to act for p_t. Owning a DStar category corresponds to acting for p_o. Having a category in a tracking label corresponds to acting for p_t, except that DStar has no external labeled input or output channels; all files and communications are implemented inside the model.

HiStar and DStar do not provide disjunctive reader policies or conjunctive writer policies, though they can be implemented by using a separate category to represent the entire policy and explicitly granting its ownership as appropriate.

Secure program partitioning [58] is largely complementary to DStar, automatically partitioning a single program into multiple programs, running on a set of machines specified at compile time with varying trust, to uphold an overall information flow policy. DStar provides mechanisms to enforce an overall information flow policy without restricting program structure, language, or partitioning mechanism. Given the sub-programs generated by secure program partitioning, DStar could potentially be used to execute them in a distributed system without trusting the compiler to correctly partition the code. Secure program partitioning has a much larger TCB, and relies on trusted external inputs to avoid a number of difficult issues addressed by DStar, such as determining when it is safe to connect to a given host at runtime, resource allocation issues, and bootstrapping.

Language-based approaches to information flow control offer some techniques for analyzing concurrent programs [44], largely restricted to static, compile-time verification. DStar checks labels at runtime,

allowing complex security policies to be defined by application code, at the cost of deferring label mismatch errors until execution time. Language-based techniques largely avoid addressing many practical issues solved by DStar, such as trust management, resource allocation, support for heterogeneous systems, and execution of arbitrary machine code.

5.5 Digital Rights Management

The problem of specifying and enforcing data security policies has received significant attention in the industry, under the name of digital rights management (DRM) in the consumer space, and information rights management (IRM) in the enterprise environment. Rights management systems sign and encrypt all sensitive data when it is written to disk or sent over the network, and only decrypt the data when handling it in memory, thereby reducing the security problem to the distribution of keys. Although an attacker without the appropriate key may not be able to decrypt the data, other information about the data, such as its size, or when it was sent, constitutes a covert channel that HiStar and DStar try to avoid.

In general, rights management systems available today tend to have orders of magnitude more trusted code than HiStar or DStar. These systems also tend to have a higher-level policy language than HiStar and DStar labels. For example, a rights management policy may allow or prohibit printing the contents of a particular document. In HiStar, such a high-level policy would be implemented by partially-trusted code with privilege to declassify certain information when sending it to the printer. The underlying kernel would then ensure that untrusted applications cannot export data via the printer, except through this partially-trusted daemon.

Consumer DRM systems are typically concerned with preventing a malicious user from getting access to certain data, while allowing trusted software on the user's machine to access the data. The application software that has access to the sensitive data is usually fully trusted not to reveal the data directly to the user. For example, the TCPA Trusted Platform Module [54] provides tamper-resistant, verifiable execution of trusted software on hardware controlled by an untrusted user. HiStar addresses a complementary problem, namely, how to minimize the amount of trust placed in software, given fully trusted hardware. DStar could use TCPA to verify whether a remote machine is running HiStar, in which case it may trust the remote machine to enforce certain information flow control restrictions.

Enterprise rights management systems, such as [11, 25, 34, 38], focus on a slightly different problem. Namely, they try to make sure that security policies are always up-to-date and that users do not accidentally reveal sensitive information to unauthorized parties, and tend to worry less about security in

the face of physical hardware access. They can track information flow between applications through the file system, clipboard, and even the display. However, these information flow tracking mechanisms are implemented by interposing on existing, high-level interfaces, and are sometimes application-specific. As a result, the amount of trusted code tracking information flow is orders of magnitude greater than that of HiStar, and such rights management systems contain numerous covert channels that HiStar closes.

In a distributed setting, enterprise rights management systems rely on a central policy server to specify data security policies. As a result, it becomes difficult for users from multiple organizations to work together on a common document, without placing the document under the control of one particular organization's policy server. On the other hand, DStar decouples the specification of an information flow control policy associated with certain data from the trust in machines that enforce this policy.

Chapter 6

Conclusion

This dissertation showed that information flow control can be used to build secure, scalable, distributed systems from largely untrusted code.

To be able to execute untrusted code with access to confidential data on a single machine, we built a new operating system called HiStar that can safely execute untrusted code, by providing strict information flow control without superuser privilege. HiStar's narrow interfaces allow for a small trusted kernel of less than 20,000 lines, on which a Unix-like environment is implemented mostly as untrusted user-level library code. By providing a shared-memory interface, HiStar is able to enforce one-way information flow control purely in the kernel, without the need for trusted user-level servers. A new container abstraction lets administrators manage and revoke resources for processes they cannot observe. Through its use of immutable labels, HiStar demonstrated that an operating system can dynamically track information flow through tainting without the taint mechanism itself leaking information. Side-by-side with the Unix environment, the system supports a number of high-security, privilege-separated applications previously not possible in a traditional Unix system.

We showed that information flow control can also be used to enforce security of untrusted code in large-scale distributed systems of multiple machines, by developing DStar, a framework for enforcing information flow control in distributed systems. The decentralized design of DStar not only improved security, but also removed any inherent scalability bottlenecks and simplified communication across administrative domains. Using self-certifying categories, DStar solved a number of difficult problems in a decentralized setting, such as determining when it is safe to communicate with a remote machine and controlling resource allocation, without introducing additional covert channels.

Using HiStar and DStar, we built a highly privilege-separated distributed web server with strong security guarantees, where most of the server code is untrusted. In most cases, even a fully compromised web server machine cannot violate the security of all users. Benchmarks showed that the web server has good overall performance and scales well with the number of physical machines. Finally, we showed that DStar allows for incremental deployment, by executing just the untrusted perl code on HiStar.

Appendix A

HiStar System Call Interface

This appendix shows in detail the system call interface provided by the HiStar kernel, in C language syntax. Sections A.1 through A.6 discuss the data types used in the system call interface, and A.7 presents the function prototypes for every system call in HiStar.

A.1 Labels

The system call interface describes Asbestos labels using a `struct ulabel`, defined as follows:

```
struct ulabel {
    uint32_t ul_size;
    uint32_t ul_nent;

    uint8_t ul_default;
    uint32_t ul_needed;

    uint64_t *ul_ent;
};
```

Category-level mappings comprising the label are stored in a contiguous array of 64-bit values pointed to by `ul_ent`. The category name, a 61-bit integer, is stored in the low bits of the 64-bit entry, and the level is stored in the upper 3 bits. Levels 0 through 3 are represented by their respective integer values, whereas level $\star$ is represented by the value 4. The default level of a label is specified by `ul_default`.

The size of the `ul_ent` array is defined by `ul_size`. `ul_nent` defines the number of used entries in the array, in case the kernel returns fewer entries than the user pre-allocated space for. Conversely, if the

user provided too few entries, the kernel sets `ul_needed` to the number of additional entries needed in `ul_ent` to represent the label.

A.2 Kernel Objects

The type of a kernel object is represented by a `kobject_type_t`:

```
typedef enum {
    kobj_container,
    kobj_thread,
    kobj_gate,
    kobj_segment,
    kobj_address_space,
    kobj_netdev
} kobject_type_t;
```

Container entries are represented by a `struct cobj_ref`:

```
struct cobj_ref {
    uint64_t container;
    uint64_t object;
};
```

A.3 Network Devices

The network device interface in HiStar allows the user to supply receive and transmit buffers for the network card. Each buffer must reside in a single memory page, to simplify kernel code. `netbuf_type` specifies either a receive or a transmit buffer:

```
typedef enum {
    netbuf_rx,
    netbuf_tx
} netbuf_type;
```

This buffer consists of a `struct netbuf_hdr`, followed by the actual data payload:

```
struct netbuf_hdr {
    uint16_t size;
    uint16_t actual_count;
};

#define NETHDR_COUNT_DONE       0x8000
#define NETHDR_COUNT_ERR        0x4000
#define NETHDR_COUNT_MASK       0x0fff
```

The `size` field, set by user-space code, specifies the size of the packet buffer, in bytes, following the `netbuf_hdr`. The `actual_count` field, on the other hand, is set by the kernel, and contains the status of the packet buffer. The lower 12 bits specify the length of the received packet, for receive buffers. The high two bits of `actual_count` specify whether the kernel is done with the buffer (`NETHDR_COUNT_DONE`) and whether any error occurred while sending or receiving the packet (`NETBUF_COUNT_ERR`).

A.4 Thread Entry Point

The initial state of a thread's execution is defined by a `struct thread_entry`:

```
enum { thread_entry_narg = 6 };

struct thread_entry {
    struct cobj_ref te_as;
    void *te_entry;
    void *te_stack;

    uint64_t te_arg[thread_entry_narg];
};
```

The `te_as` field specifies the container entry of the thread's address space. `te_entry` and `te_stack` specify the initial program counter and stack pointer. 6 additional 64-bit arguments are passed to the entry point in `te_arg`. On the 64-bit x86-64 processors, these arguments are copied directly into architectural registers. On other platforms, such as the 32-bit i386, these arguments are stored in the thread object, and made accessible to the application code through the `sys_self_get_entry_args` system call, which returns a `struct thread_entry_args`:

```
struct thread_entry_args {
    uint64_t te_arg[thread_entry_narg];
};
```

As an optimization, truncated 32-bit values of the first 3 arguments are passed in architectural registers when running on the 32-bit i386.

A.5 Address Space

An address space is described using a `struct u_address_space`:

```
struct u_address_space {
    void *trap_handler;
    void *trap_stack_base;
    void *trap_stack_top;
    uint64_t size;
    uint64_t nent;
    struct u_segment_mapping *ents;
};
```

Mappings between virtual addresses and segments are stored in the `ents` array. `size` and `nent` specify the number of available and used mapping entries, respectively. The `trap_handler`, `trap_stack_base`, and `trap_stack_top` fields specify how to handle an alert sent to a thread in this address space.

Each mapping is represented by a `struct u_segment_mapping`:

```
struct u_segment_mapping {
    struct cobj_ref segment;
    uint64_t start_page;
    uint64_t num_pages;
    uint32_t kslot;
    uint32_t flags;
    void *va;
};

#define SEGMAP_EXEC             0x01
#define SEGMAP_WRITE            0x02
#define SEGMAP_READ             0x04
#define SEGMAP_REVERSE_PAGES    0x08
```

`segment` specifies a container entry for the segment being mapped. `start_page` and `num_pages` specify what subset of the segment's pages should be mapped. `kslot` is a kernel-internal identifier that allows a specific segment mapping to be changed later on without changing the entire address space. `va` specifies the virtual address at which the specified sequence of pages should be mapped.

Flags controlling the segment mapping are specified in `flags`. These include permission bits (read, write, and execute), as well as an optimization for handling grow-down stacks in the x86 architecture: the `SEGMAP_REVERSE_PAGES` flag reverses the order of pages in the mapping.

A.6 Return Values

Most system call functions return a signed integer value to signify the success or failure of an operation. Operations that allocate objects return an `int64_t`, which, if negative, specifies an error code, and if positive, specifies the object ID of the newly created object. The container, to form a container entry for the new object, is specified when allocating the object.

The possible error code values (returned as negative numbers in case of an error) are as follows:

```
enum {
    E_UNSPEC = 1,       // Unspecified or unknown problem
    E_INVAL,            // Invalid parameter
    E_NO_MEM,           // Request failed due to memory shortage
    E_RESTART,          // Restart system call
    E_NOT_FOUND,        // Object not found
    E_LABEL,            // label check error
    E_BUSY,             // device busy
    E_NO_SPACE,         // not enough space in buffer
    E_AGAIN,            // try again
    E_IO,               // disk IO error
    E_FIXED_QUOTA,      // object has a fixed quota
    E_VAR_QUOTA,        // object has a variable quota
    E_RESOURCE          // container out of space
};
```

A.7 System Call Function Prototypes

A.7.1 Console

```
int     sys_cons_puts(const char *s, uint64_t size);
int     sys_cons_getc(void);
int     sys_cons_probe(void);
```

A.7.2 Objects

```
int     sys_obj_unref(struct cobj_ref o);
kobject_type_t
        sys_obj_get_type(struct cobj_ref o);
int     sys_obj_get_label(struct cobj_ref o, struct ulabel *l);
int     sys_obj_get_name(struct cobj_ref o, char *name);
int64_t sys_obj_get_quota_total(struct cobj_ref o);
int64_t sys_obj_get_quota_avail(struct cobj_ref o);
int     sys_obj_get_meta(struct cobj_ref o, void *meta);
int     sys_obj_set_meta(struct cobj_ref o,
                         const void *oldm, void *newm);
int     sys_obj_set_fixedquota(struct cobj_ref o);
int     sys_obj_set_readonly(struct cobj_ref o);
int     sys_obj_get_readonly(struct cobj_ref o);
```

A.7.3 Network Devices

```
int64_t sys_net_create(uint64_t container, uint64_t card_idx,
                       const struct ulabel *l, const char *name);
int64_t sys_net_wait(struct cobj_ref ndev, uint64_t waiter_id,
                     int64_t waitgen);
int     sys_net_buf(struct cobj_ref ndev, struct cobj_ref seg,
                    uint64_t offset, netbuf_type type);
int     sys_net_macaddr(struct cobj_ref ndev, uint8_t *buf);
```

A.7.4 Containers

```
int64_t sys_container_alloc(uint64_t parent, const struct ulabel *l,
                            const char *name, uint64_t avoid_types,
                            uint64_t quota);
int64_t sys_container_get_nslots(uint64_t container);
int64_t sys_container_get_parent(uint64_t container);
int64_t sys_container_get_slot_id(uint64_t container, uint64_t slot);
int     sys_container_move_quota(uint64_t parent, uint64_t child,
                                 int64_t nbytes);
```

A.7.5 Gates

```
int64_t sys_gate_create(uint64_t container,
```

```
                        const struct thread_entry *s,
                        const struct ulabel *label,
                        const struct ulabel *clear,
                        const struct ulabel *verify,
                        const char *name, int entry_visible);
int     sys_gate_enter(struct cobj_ref gate,
                       const struct ulabel *label,
                       const struct ulabel *clearance,
                       const struct thread_entry *s);
int     sys_gate_clearance(struct cobj_ref gate, struct ulabel *ul);
int     sys_gate_get_entry(struct cobj_ref gate,
                           struct thread_entry *s);
```

A.7.6 Segments

```
int64_t sys_segment_create(uint64_t container, uint64_t num_bytes,
                           const struct ulabel *l, const char *name);
int64_t sys_segment_copy(struct cobj_ref seg, uint64_t container,
                         const struct ulabel *l, const char *name);
int     sys_segment_addref(struct cobj_ref seg, uint64_t ct);
int     sys_segment_resize(struct cobj_ref seg, uint64_t num_bytes);
int64_t sys_segment_get_nbytes(struct cobj_ref seg);
int     sys_segment_sync(struct cobj_ref seg, uint64_t start,
                         uint64_t nbytes, uint64_t pstate_ts);
```

A.7.7 Address Spaces

```
int64_t sys_as_create(uint64_t container, const struct ulabel *l,
                      const char *name);
int64_t sys_as_copy(struct cobj_ref as, uint64_t container,
                    const struct ulabel *l, const char *name);
int     sys_as_get(struct cobj_ref as, struct u_address_space *uas);
int     sys_as_set(struct cobj_ref as, struct u_address_space *uas);
int     sys_as_get_slot(struct cobj_ref as,
                        struct u_segment_mapping *usm);
int     sys_as_set_slot(struct cobj_ref as,
                        struct u_segment_mapping *usm);
```

A.7.8 Threads

```
int64_t sys_thread_create(uint64_t container, const char *name);
```

```
int     sys_thread_start(struct cobj_ref thread,
                         const struct thread_entry *s,
                         const struct ulabel *l,
                         const struct ulabel *clear);
int     sys_thread_trap(struct cobj_ref thread, struct cobj_ref as,
                        uint32_t trapno, uint64_t arg);
```

A.7.9 Thread acting on itself

```
void    sys_self_yield(void);
void    sys_self_halt(void);
int64_t sys_self_id(void);
int     sys_self_addref(uint64_t container);
int     sys_self_get_as(struct cobj_ref *as_obj);
int     sys_self_set_as(struct cobj_ref as_obj);
int     sys_self_set_label(const struct ulabel *l);
int     sys_self_set_clearance(const struct ulabel *l);
int     sys_self_get_clearance(struct ulabel *l);
int     sys_self_set_verify(const struct ulabel *l,
                            const struct ulabel *c);
int     sys_self_get_verify(struct ulabel *l, struct ulabel *c);
int     sys_self_fp_enable(void);
int     sys_self_fp_disable(void);
int     sys_self_set_waitslots(uint64_t nslots);
int     sys_self_set_sched_parents(uint64_t p0, uint64_t p1);
int     sys_self_set_cflush(int cflush);
int     sys_self_get_entry_args(struct thread_entry_args *targ);
```

A.7.10 Sleep and wakeup

```
int     sys_sync_wait(volatile uint64_t *addr, uint64_t val,
                      uint64_t wakeup_at_nsec);
int     sys_sync_wait_multi(volatile uint64_t **addrs, uint64_t *vals,
                            uint64_t num, uint64_t nsec);
int     sys_sync_wakeup(volatile uint64_t *addr);
```

A.7.11 Miscellaneous

```
int64_t sys_handle_create(void);

int     sys_machine_reboot(void);
```

```
int64_t sys_clock_nsec(void);

int64_t sys_pstate_timestamp(void);
int     sys_pstate_sync(uint64_t timestamp);
```

Appendix B

DStar Network Protocol

This appendix gives a precise definition of the DStar network protocol discussed in Chapter 3, using the XDR protocol description language [50]. Parts of the protocol for handling public key encryption and signatures are borrowed from SFS [28], in particular the `sfs_pubkey2`, `sfs_sig2`, and `sfs_ctext2` encodings.

B.1 Protocol Definition

```
/*
 * Distributed HiStar protocol
 */

%#include <sfs_prot.h>

typedef unsigned dj_timestamp;  /* UNIX seconds */
typedef opaque dj_stmt_blob<>;  /* No recursive definitions in XDR */
typedef sfs_pubkey2 dj_pubkey;
typedef sfs_sig2 dj_sign;

struct dj_gcat {                /* Global category name */
    dj_pubkey key;
    unsigned hyper id;
    bool integrity;
};

struct dj_address {
```

```
    unsigned ip;                /* network byte order */
    unsigned port;              /* network byte order */
};

/*
 * Labels.
 */

struct dj_label {
    dj_gcat ents<>;
};

struct dj_cat_mapping {
    dj_gcat gcat;
    unsigned hyper lcat;

    unsigned hyper res_ct;      /* sub-ct storing the mapping */
    unsigned hyper res_gt;      /* unbound gate providing { lcat* } */
};

struct dj_catmap {
    dj_cat_mapping ents<>;
};

/*
 * Delegations.
 */

enum dj_entity_type {
    ENT_PUBKEY = 1,
    ENT_GCAT,
    ENT_ADDRESS
};

union dj_entity switch (dj_entity_type type) {
 case ENT_PUBKEY:
    dj_pubkey key;
 case ENT_GCAT:
    dj_gcat gcat;
 case ENT_ADDRESS:
    dj_address addr;
};

struct dj_delegation {          /* via says a speaks-for b */
    dj_entity a;
    dj_entity b;
    dj_pubkey *via;
```

```
    dj_timestamp from_ts;
    dj_timestamp until_ts;
};

struct dj_delegation_set {
    dj_stmt_blob ents<>;        /* XDR-encoded dj_signed_stmt */
};

/*
 * Message transfer.
 */

enum dj_slot_type {
    EP_GATE = 1,
    EP_SEGMENT,
    EP_MAPCREATE,
    EP_DELEGATOR
};

enum dj_special_gate_ids {      /* set container to zero */
    GSPEC_CTALLOC = 1,
    GSPEC_ECHO,
    GSPEC_GUARDCALL
};

struct dj_gatename {
    unsigned hyper gate_ct;
    unsigned hyper gate_id;
};

struct dj_ep_gate {
    unsigned hyper msg_ct;      /* container for segment & thread */
    dj_gatename gate;
};

struct dj_ep_segment {
    unsigned hyper seg_ct;
    unsigned hyper seg_id;
};

union dj_slot switch (dj_slot_type type) {
 case EP_GATE:
    dj_ep_gate ep_gate;
 case EP_SEGMENT:
    dj_ep_segment ep_segment;
 case EP_MAPCREATE:
    void;
```

```
 case EP_DELEGATOR:
    void;
};

struct dj_message {
    dj_pubkey from;
    dj_pubkey to;

    dj_slot target;             /* gate or segment to deliver to */
    dj_label tracking;          /* tracking label */
    dj_label glabel;            /* grant label for gates */
    dj_label gclear;            /* grant clearance for gates */
    dj_catmap catmap;           /* target node category mappings */
    dj_delegation_set dset;     /* supporting delegations */
    opaque msg<>;
};

/*
 * Not all message delivery codes are exposed to the sender.
 */

enum dj_delivery_code {
    DELIVERY_DONE = 1,
    DELIVERY_TIMEOUT,
    DELIVERY_NO_ADDRESS,
    DELIVERY_LOCAL_DELEGATION,
    DELIVERY_REMOTE_DELEGATION,
    DELIVERY_LOCAL_MAPPING,
    DELIVERY_REMOTE_MAPPING,
    DELIVERY_LOCAL_ERR,
    DELIVERY_REMOTE_ERR
};

/*
 * Session key establishment.
 */

struct dj_key_setup {
    dj_pubkey sender;
    dj_pubkey to;
    sfs_ctext2 kmsg;
};

/*
 * Signed statements that can be made by entities.  Every network
 * message is a statement.
 *
```

```
 * Fully self-describing statements ensure that one statement
 * cannot be mistaken for another in a different context.
 */

enum dj_stmt_type {
    STMT_DELEGATION = 1,
    STMT_MSG,
    STMT_KEY_SETUP
};

union dj_stmt switch (dj_stmt_type type) {
 case STMT_DELEGATION:
    dj_delegation delegation;
 case STMT_MSG:
    dj_message msg;
 case STMT_KEY_SETUP:
    dj_key_setup keysetup;
};

struct dj_stmt_signed {
    dj_stmt stmt;
    dj_sign sign;
};
```

Bibliography

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006. `http://eprint.iacr.org/`.

[2] James P. Anderson. A unification of computer and network security concepts. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 77–87, Oakland, CA, 1985.

[3] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[4] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[5] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, Washington, DC, August 2003.

[6] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *Proceedings of the 15th USENIX Security Symposium*, pages 137–151, Vancouver, BC, 2006.

[7] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 2007 USENIX Security Symposium*, pages 1–16, Boston, MA, August 2007.

[8] ClamAV. `http://www.clamav.net/`.

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

[10] 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 *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 2005.

[11] EMC Documentum Information Rights Management. `http://software.emc.com/products/product_family/documentum_family.htm`.

[12] Deborah Estrin. Non-discretionary controls for inter-organization networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–61, Oakland, CA, 1985.

[13] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, pages 479–495, Ottawa, Canada, June 2002.

[14] 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.

[15] Sudhakar Govindavajhala and Andrew W. Appel. Windows access control demystified. Technical Report TR-744-06, Princeton University, Department of Computer Science, January 2006.

[16] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the Summer 1993 USENIX Technical Conference*, pages 147–159, April 1993.

[17] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, October 1988.

[18] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Mnuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.

[19] James W. Gray III. On introducing noise into the bus-contention channel. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 90–98, Oakland, CA, May 1993.

[20] International Organization for Standardization. Information technology–portable operating system interface (POSIX), 1003.1-2004. Section 4.4: File Access Permissions.

[21] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory per-packet access control. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, Baltimore, MD, August 2006.

[22] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4):263–297, November 2000.

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

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

[25] Liquid Machines. `http://www.liquidmachines.com/`.

[26] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Technical Conference*, pages 29–40, June 2001. FREENIX track.

[27] lwIP. `http://savannah.nongnu.org/projects/lwip/`.

[28] David Mazières. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology, May 2000.

[29] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, pages 261–274, June 2001.

[30] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.

[31] John McHugh and Andrew P. Moore. A security policy and formal top-level specification for a multi-level secure local area network. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 34–39, Oakland, CA, 1986.

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

[33] Microsoft Corporation. Access rights and access masks. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_rights_and_access_masks.asp.

[34] Microsoft Windows Server 2003 Rights Management Services. http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/.

[35] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[36] Ryan Naraine. Symantec antivirus worm hole puts millions at risk. *eWeek.com*, May 2006. http://www.eweek.com/article2/0,1895,1967941,00.asp.

[37] OpenVPN. http://openvpn.net/.

[38] Oracle Information Rights Management. http://www.oracle.com/products/middleware/content-management/information-rights-management.html.

[39] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[40] Herbert Pötzl. *Linux-VServer Technology*, 2004. http://linux-vserver.org/Linux-VServer-Paper.

[41] Kevin Poulsen. Slammer worm crashed Ohio nuke plant net. *The Register*, August 20, 2003. http://www.theregister.co.uk/2003/08/20/slammer_worm_crashed_ohio_nuke/.

[42] Privacy Rights Clearinghouse. A chronology of data breaches. September 2007. http://www.privacyrights.org/ar/ChronDataBreaches.htm.

[43] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[44] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[45] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. In *Proceedings of the Third Symposium on Operating Systems Principles*, pages 42–54, March 1972.

[46] Seagate. *Barracuda 7200.7 Product Manual*, Publication 100217279, Rev. L edition, March 2004. `http://www.seagate.com/support/disc/manuals/ata/cuda7200pm.pdf`.

[47] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[48] Deepinder P. Sidhu and Morrie Gasser. A multilevel secure local area network. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–143, Oakland, CA, 1982.

[49] Phil Snowberger and Douglas Thain. Sub-identities: Towards operating system support for distributed system security. Technical Report 2005-18, University of Notre Dame, Department of Computer Science and Engineering, October 2005.

[50] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[51] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, January 1996.

[52] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, December 1990.

[53] Think Computer Corporation. Identity crisis. `http://www.thinkcomputer.com/corporate/whitepapers/identitycrisis.pdf`.

[54] Trusted Computing Group's Trusted Platform Module. `https://www.trustedcomputinggroup.org/groups/tpm/`.

[55] uClibc. `http://uclibc.org/`.

[56] Edward P. Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[57] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[58] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 1–14, October 2001.

[59] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.