MIT Libraries | DSpace@MIT

MIT Open Access Articles

Making information flow explicit in HiStar

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2011. Making information flow explicit in HiStar. Commun. ACM 54, 11 (November 2011), 93-101.

As Published: http://dx.doi.org/10.1145/2018396.2018419

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/73665

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0

Massachusetts Institute of Technology

Making Information Flow Explicit in HiStar

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières
Stanford and UCLA

Abstract

HiStar is a new operating system designed to minimize the amount of code that must be trusted. HiStar provides strict information flow control, which allows users to specify precise data security policies without unduly limiting the structure of applications. HiStar's security features make it possible to implement a Unix-like environment with acceptable performance almost entirely in an untrusted user-level library. 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.

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, the security of most systems ultimately depends on a large fraction of the code behaving correctly. Unfortunately, experience has 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. As a result, we see a steady stream of high-profile security incidents.

How can we build secure systems when we cannot trust programmers to write secure code? One hope is to separate the security critical portions of an application from the untrusted bulk of its implementation; if security depends on only 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. Unfortunately, traditional operating systems do not lend themselves to such a division of functionality; they make it too difficult to predict the full implications of every action by untrusted code. HiStar is a new operating system designed to overcome this limitation.

HiStar enforces security by controlling how information flows through the system. Hence, one can reason about which components of a system may affect which others and how, without having to understand those components themselves. Specifying policies in terms of information flow is often much easier than reasoning about the security implications of individual operations.

As an example, consider the recently discovered critical vulnerability in Norton Antivirus that put millions of systems at risk of remote compromise [15]. 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 to enforce security without trusting ClamAV, thereby minimizing potential damage from ClamAV's vulnerabilities.

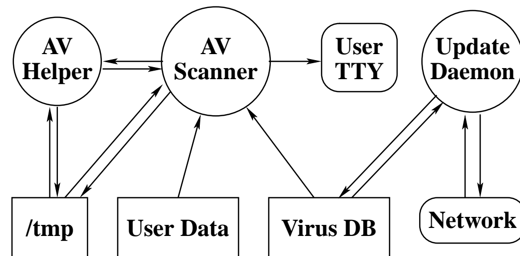

Figure 1: The ClamAV virus scanner. Circles represent processes, rectangles represent files and directories, and rounded rectangles represent devices. Arrows represent the expected data flow for a well-behaved virus scanner.

Figure 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 programs to decode input files. Here are just a 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 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 shows our port of ClamAV to HiStar. There are two differences from Linux. First, we have labeled files with private user data as *tainted*. Tainting a file restricts the flow of its contents to any untainted component, including the network. A file can be labeled with arbitrarily many *categories* of taint. Whoever allocates a category—in this case the file owner—has the exclusive ability to *untaint* data in that category.

The second difference from Linux is that we have launched the scanner from a new, 110-line program called *wrap*, to which we give untainting privileges. *wrap* untaints the virus scanner's result and reports back to the user. The scanner cannot read tainted user files without first tainting itself. Once tainted, it can no longer convey information to the network or update daemon. So long as *wrap* is correctly implemented, then, ClamAV cannot leak the contents of the files it scans.

Though this paper will use the virus scanner as a running example, a number of other typical security problems can more easily be couched in terms of information flow. For example, protecting users' private profile information on a web site often boils down to ensuring one person's information (social security number, credit card, etc.) cannot be sent to another user's browser. Protecting against trojan horses means ensuring network payloads do not affect the contents of system

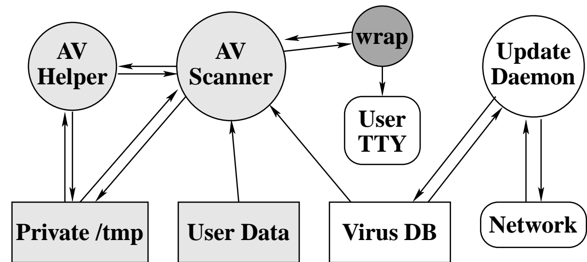

Figure 2: ClamAV running in HiStar. Lightly-shaded components are *tainted*, which prevents them from conveying any information to untainted (unshaded) components. The strongly-shaded `wrap` has untainting privileges, allowing it to relay the scanner's output to the terminal.

files. Protecting passwords means ensuring that whatever code verifies them can reveal only the single bit signifying whether or not authentication succeeded. HiStar provides a new, Unix-like development environment in which small amounts of code can secure much larger, untrusted applications by enforcing such policies.

The information flow principles behind this type of isolation are not new. Mechanisms in several other systems, including SELinux [11], EROS [23], and Asbestos [5], are also capable of isolating an untrusted virus scanner. HiStar's taint labels, which originated in Asbestos, have features resembling the language-based labels in Jif and Jflow [14]. 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 tainting without the taint mechanism itself leaking information. 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 with no special right to untaint, read, or write arbitrary user data.

2 Labels

HiStar tracks and enforces information flow using Asbestos *labels* [5]. All operating system abstractions are layered on top of six low-level kernel object types described in the next section—threads, address spaces, segments, gates, containers, and devices. Every object has a label. The label specifies, for each category of taint, whether the object has untainting privileges for that category (threads and gates can have such privileges), and, if not, how tainted the object is in that category. Any system call or page fault can cause information to flow between the current thread and other objects. However, the kernel disallows actions that would convey information from more to less tainted objects in any given category.

A label is a function from categories to taint *levels*.

Level	Meaning in an object's label
$\star$	has untainting privileges in this category
0	cannot be written/modified by default
1	default level—no restriction in this category
2	cannot be untainted/exported by default
3	cannot be read/observed by default

Figure 3: An object's label assigns it one of the above taint levels in each category. Only thread and gate labels may contain $\star$.

Any given label maps all but a small number of categories to some default background taint level for the object—usually **1**. Thus, a label consists of a default taint level and a list of categories in which the object is either more or less tainted 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 $L = \{w\mathbf{0},\, r\mathbf{3},\, \mathbf{1}\}$, which is just a more compact way of designating the function

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

Each category in which an object's taint 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 label $L_T = \{\mathbf{1}\}$, and an object O with label $L_O = \{c\mathbf{3},\, \mathbf{1}\}$. Because $L_T(c) = \mathbf{1} < \mathbf{3} = L_O(c)$, O is more tainted than T in category c. Hence, no information may flow from O to T, which means the thread cannot read or observe the object. Conversely, an object may be less tainted than the default. If instead an object O' has $L_{O'} = \{c\mathbf{0},\, \mathbf{1}\}$, then $L_{O'}(c) = \mathbf{0} < \mathbf{1} = L_T(c)$, and no information can flow from T to O', meaning the thread cannot write to or modify the object.

Any given category in an object's label restricts either reading or writing the object, but not both. (It is, of course, common to restrict both by using two categories.) While conventional operating systems can either permit or prohibit read access to an object such as a file, HiStar allows a third option: permit a thread to read an object so long as it does not untaint the data or export it from the machine. In some cases, such as VPN isolation discussed in Section 6.3, it is convenient to make read without untainting the default permission for a given category. Therefore, HiStar supports two levels more tainted than the default: **2** and **3**. The difference arises because threads may chose to taint themselves to read more tainted objects, but only up to another label called their *clearance*, which defaults to $\{\mathbf{2}\}$.

The final taint level is $\star$ ("Star"). It signifies untainting privileges within a category, and may appear only in a thread or gate label. Roughly speaking, when a thread is at level $\star$ in a particular category, the kernel ignores that category in performing label checks for operations by that thread. In other words, if a thread T with label L_T has $L_T(c) = \star$, the thread can bypass information flow restrictions in c; 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. Figure 3 summarizes taint levels that appear in object labels.

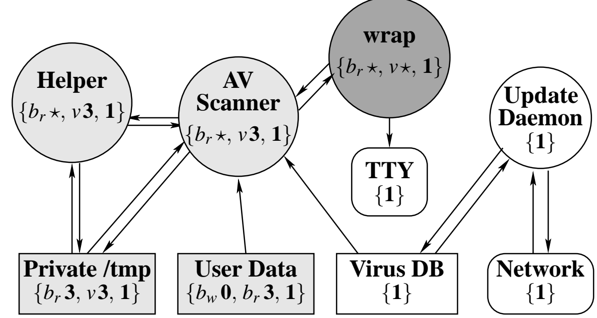

Figure 4: Labels on components of the HiStar ClamAV port.

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. 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. (The specific length 61 was chosen to fit a category name and 3-bit taint level in the same 64-bit field, which facilitated the label implementation.)

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 that must be assigned by the privileged security administrator.

2.1 Example

Returning to the virus scanner example, Figure 4 shows a simplified version of the 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, the update daemon could directly read Bob's files and transmit them over the network. In Unix, Bob's files would be protected either by setting file permission bits to 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 is for Bob to allocate two categories, b_r and b_w, to restrict read and write access to his 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 b_r acts like a read capability. Similarly b_w acts like a write capability. The authentication mechanism

described in Section 6.2 grants Bob's shell ownership of the two categories whenever he logs in.

The *wrap* program is invoked with all of Bob's privileges—in particular with ownership of b_r, the category that restricts read access to Bob's files. *wrap* allocates a new category, v, to isolate the scanner, creates a private `/tmp` directory writable at taint level **3** in category v, then launches the scanner tainted **3** in category v. The v taint prevents the scanner, or any process it creates, from communicating to the update daemon or network, except through wrap (which has untainting privileges in v). The v taint also prevents the scanner, or any program it spawns, from modifying any of Bob's files, because those files are all less tainted (at the default level of **1**) in category v.

2.2 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 tainted as L_1 in every category. This relationship is so important that we introduce a symbol, $\sqsubseteq$, to denote it:

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

Level $\star$ complicates matters since it represents ownership and untainting privileges rather than taint. A thread T whose 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 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 tainted 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 taint level in the category. This shift from high to low implements untainting.

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 untainting is denoted by superscript operators $^{✪}$ and * that translate $\star$ to ✪ and ✪ to $\star$, respectively. For example, if $L = \{a\star, b✪, \mathbf{1}\}$, then $L^{✪} = \{a✪, b✪, \mathbf{1}\}$ 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 $L_O \sqsubseteq L_T^{✪}$ (i.e., "no read up").
- T can modify O, which in HiStar implies observing O, only if $L_T \sqsubseteq L_O \sqsubseteq L_T^{✪}$ (i.e., "no write down").

These two basic conditions appear repeatedly in our description of HiStar's abstractions.

Labels form a lattice [4] under the partial order of the $\sqsubseteq$ relation. 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 taint themselves to observe more tainted objects. To observe an object O labeled L_O, a thread T labeled L_T must raise its label to at least $L'_T = (L_T^{✪} \sqcup L_O)^{\star}$, because that is the lowest label satisfying both $L_T \sqsubseteq L'_T$ and $L_O \sqsubseteq L_T'^{✪}$.

3 Kernel design

As previously mentioned, the HiStar kernel is organized around six object types. Every object has a unique, 61-bit *object ID*, a *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' labels are specified at creation and then immutable. Some objects allow efficient copies to be made with different labels, which is useful in cases that might otherwise require re-labeling.

An object's 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 is more tainted 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.

To revisit the virus scanner example, suppose data from the scanner, tainted $v\mathbf{3}$, was somehow observed by the update daemon, with a 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.

Unfortunately, information flow control is not perfect. Tainted malicious software can leak information through

covert channels—for instance, by modulating CPU usage in a way that affects the response time of untainted threads. A related problem is preventing malicious software from making even properly tainted 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 tainted 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 label and on the labels of objects the thread allocates or grants storage to. In the virus scanner example, the update daemon cannot read Bob's private files, labeled $\{b_r\mathbf{3}, b_w\mathbf{0}, \mathbf{1}\}$, because its clearance of $\{\mathbf{2}\}$ prevents it from tainting itself $b_r\mathbf{3}$.

HiStar has a single-level store—on bootup, the entire system state is restored from the most recent on-disk snapshot. This eliminates the need for trusted boot scripts to re-initialize processes such as daemons that on more traditional operating systems would not survive a reboot. It also achieves economy of mechanism by allowing the file system to be implemented with the same kernel abstractions as virtual memory. 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 is a segment, providing a variable-length byte array—similar to a file in other operating systems. The rest of this section discusses other HiStar kernel object types.

3.1 Threads

As previously mentioned, each thread T has a label L_T and a clearance 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)

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

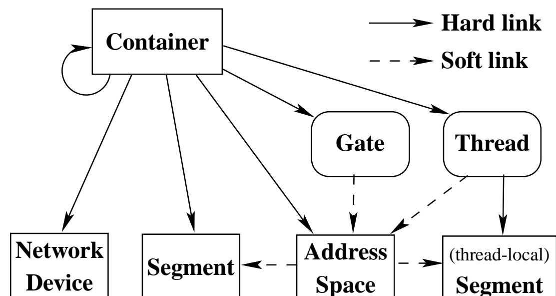

Figure 5: Kernel object types in HiStar. *Soft links* name objects by a particular ⟨container ID, object ID⟩ container entry. Threads and gates, which can own categories (i.e., contain $\star$ in their labels), are represented by rounded rectangles.

T may raise its own label through the system call

- int *self_set_label* (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 tainted object. T can also lower its clearance in any category (but not below its 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 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 $L_T \sqsubseteq L_{T'} \sqsubseteq C_{T'} \sqsubseteq C_T$.

3.2 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. We must ensure such resources can nonetheless be deallocated.

HiStar provides hierarchical control over object allocation and deallocation through a *container* abstraction. Like Unix directories, containers hold *hard links* to objects. There is a specially-designated root container, which can never be deallocated. Any other object is deallocated once there is no path to it from the root container. Figure 5 shows the possible links between containers and other object types.

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 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 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 its clearance is high enough to allocate objects at S's label ($L_S \sqsubseteq C_T$). T can thus prolong S's life even without permission to modify S. A 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). Most system calls therefore specify objects not by ID, but by $\langle$container ID, object ID$\rangle$ pairs, called *container entries*. 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 $L_T \sqsubseteq L_{T'}^{\star}$, as required.

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 label. (Since thread labels are not immutable, T can only read the label of another thread T' if $L_{T'}^{\star} \sqsubseteq L_T^{\star}$.) By examining the labels of objects more tainted than themselves, threads can determine how they must taint themselves 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 $L_D \sqsubseteq L_T^{\star}$, 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 making D less tainted than its parent in one or more categories, the thread T' that created D effectively preauthorized a small amount of information to be transmitted from threads that can delete D to threads that can use D. Fortunately, the allocation rules ($L_{T'} \sqsubseteq L_{D'} \sqsubseteq L_{T'}^{\star}$ 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 $L_D(c) < L_{D'}(c)$.

3.3 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 $L_T \sqsubseteq L_D \sqsubseteq L_T^{\star}$ and $L_T \sqsubseteq L_O \sqsubseteq C_T$. If $n < 0$, L_T must also satisfy $L_O \sqsubseteq L_T^{\star}$ 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.

3.4 Address spaces

Every running 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 permission (and some convenience bits for user-level software).

Each address space A has a 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 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., $L_{T'} \sqsubseteq L_A \sqsubseteq L_{T'}^{\circledast}$) and to observe T (i.e., $L_T \sqsubseteq L_{T'}^{\circledast}$). 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'.

3.5 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, 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, to prove possession of categories without granting them across the gate call. Gate invocation is permitted when $L_T \sqsubseteq C_G$, $L_T \sqsubseteq L_V$, and $(L_T^{\circledast} \sqcup L_G^{\circledast})^{\star} \sqsubseteq L_R \sqsubseteq C_R \sqsubseteq (C_T \sqcup C_G)$. The entry point function can examine L_V for additional access control. Note that thread 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 initial resources—namely, the thread object which 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 6.2, uses gates to obtain the user's privileges. The use of gates in user-level applications is discussed in more detail in Section 5.5.

4 Kernel Implementation

Our implementation of HiStar runs on x86-64 processors, such as AMD Opteron and Athlon64 CPUs. 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 [24]. 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.

4.1 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 consists of 15,200 lines of C code (of which 5,700 lines contain a semicolon) and 150 lines of assembly; this is roughly 45% fewer lines of C code than the Asbestos kernel. The source code consists of the following rough components:

- 3,400 lines of architecture-specific code, implementing virtual memory and threads.
- 4,000 lines of code for B+-trees, write-ahead logging and object persistence.
- 3,000 lines of code for device drivers, including PCI support, DMA-based IDE, console, and three network drivers.
- 4,800 lines of code for system calls, containers, profiling, and other hardware-independent components.

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 [6] 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). When hardware support for IO virtualization becomes available, we expect to move many device drivers out of the fully-trusted kernel.

5 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 [25] to HiStar. The HiStar platform-specific code is a small layer underneath uClibc that emulates the Linux system call interface, comprising approximately 10,000 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, the links web browser and OpenSSH, in many cases requiring little or no source code modifications. The rest of this section discusses the design and implementation of our Unix emulation library.

5.1 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. 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 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 listing the files in that directory, and write privilege allows creating new files and renaming or deleting existing files.

5.2 Processes

A process in HiStar is a user-space convention. Figure 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 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 label $\{p_r\star, p_w\star, \mathbf{1}\}$ 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.

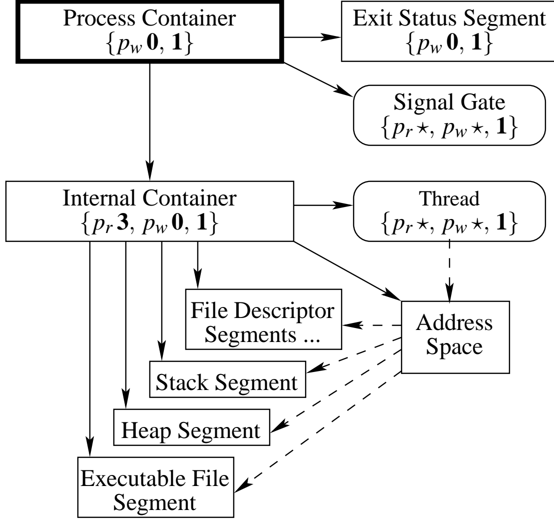

Figure 6: Structure of a HiStar process. A process container is represented by a thick border. Not shown are some label components that prevent other users from signaling the process or reading its exit status.

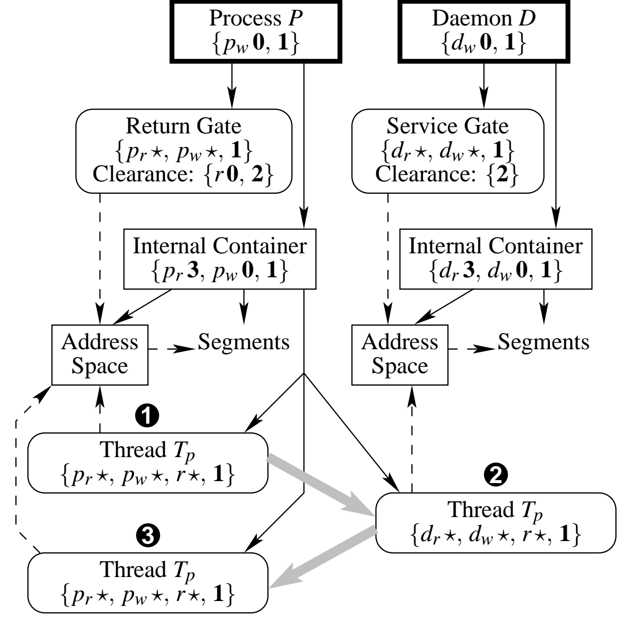

Figure 7: Objects involved in a gate call operation. Thick borders represent process containers. r is the return category; d_r and d_w are the process read and write categories for daemon D. Three states of the same thread object T_p are shown: 1) just before calling the service gate, 2) after calling the service gate, and 3) after calling the return gate.

5.3 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 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 setting a thread's label to $\{f_r\star,\,f_w\star,\,\mathbf{1}\}$. 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.

5.4 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 label containing $u_r\star$, $u_w\star$, and users' private files would have a 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. (Doing so 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 6.2.

5.5 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 *return category* r is allocated to prevent arbitrary threads from invoking the return gate; the return gate's clearance requires ownership of the return category to invoke it, and the caller grants the return category when invoking the service gate. Figure 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. A thread running with this 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. 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. Stateless services such as the timestamping daemon are usually well-suited to forking, whereas services that maintain mutable shared state may want to avoid forking by refusing tainted gate calls.

5.6 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 clearance on the signal gate is $\{u_w\,\mathbf{0},\ \mathbf{2}\}$, 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.

5.7 Networking

HiStar uses the lwIP [12] 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 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.

5.8 Explicit Information Leaks

Unix was not designed to control information flow. Emulating certain aspects therefore 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, user code 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. Of these, file creation has by far the biggest information flow, declassifying the name of the newly created file. Low-secrecy applications concerned only with accidental disclosure allow these operations. Higher-secrecy applications may choose to set fixed quotas for tainted objects and only declassify process exits. The next section shows examples of such applications.

6 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.

6.1 Anti-Virus Software

We have implemented an untrusted virus scanner, as suggested in several examples, by porting ClamAV [3] and using the *wrap* program to run it in isolation. To provide strong isolation, *wrap* does not create the standard Unix untainting gate 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.

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 ar-

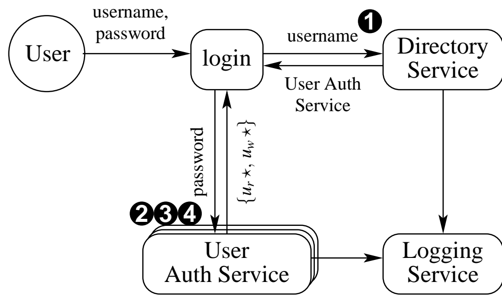

Figure 8: A high-level overview of the authentication system.

bitrary code in place of ClamAV, the label set by *wrap* when running ClamAV ensures that private information cannot be exported.

6.2 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.

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 is leaked. Providing such isolation under a traditional operating system would be difficult.

Figure 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

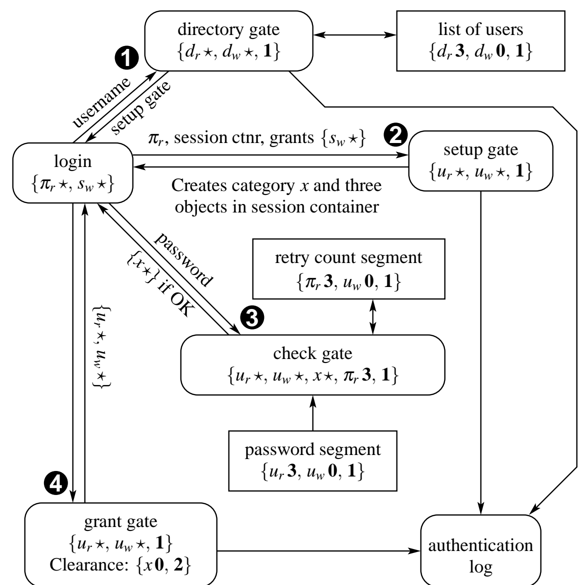

Figure 9: A detailed view of the interactions between authentication system components. The setup gate, check gate and grant gate (2, 3 and 4) are all part of the user's authentication service.

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 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 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 category, protects the password from disclosure. The s_w category controls write access to a *login session container*, which login

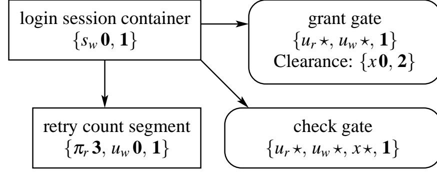

Figure 10: Objects created by the user's setup gate in the session container.

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 the session container, shown in Figure 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 comprises 58 lines of code; the directory service comprises 188 lines, and the standard password-based user authentication service comprises 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.

6.3 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 [19]. 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 6.1.

We have implemented VPN isolation around the popular OpenVPN package [16]. Figure 11 shows the components of the system and their labels: The VPN runs a second lwIP stack which talks to the OpenVPN client

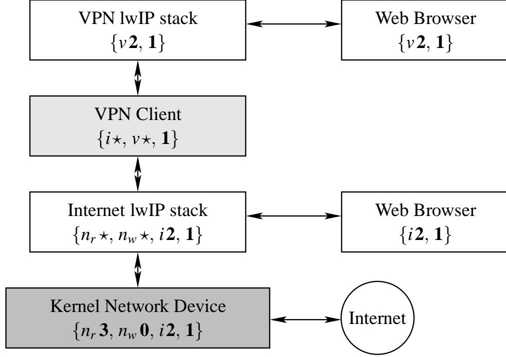

Figure 11: Secure VPN application. The VPN client is trusted to taint incoming VPN packets with $\{v\mathbf{2}\}$, reject any outgoing packets tainted in category i, and properly encrypt/decrypt data. The kernel network device is completely trusted. Neither of the lwIP stacks is trusted.

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	—

Figure 12: Microbenchmark results on HiStar, Linux and OpenBSD.

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). 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 5.8.

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.

6.4 Web Services

The original motivating application for Asbestos 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. Authentication uses an instance of the daemon described in Section 6.2. HiStar also has an experimental privilege-separated database; unlike the Asbestos database, it does not support standard SQL queries. (Whether it will prove general enough for most web services is still an open question.) Since the benefits of Asbestos-style web services have been reported elsewhere, this paper concentrates on other applications whose architecture is more unique to HiStar.

7 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* 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.

7.1 Microbenchmarks

To evaluate the performance of specific aspects of HiStar, we chose four microbenchmarks: LFS small-file and large-file benchmarks [20], 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. Figure 12 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 run-

ning 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 Figure 12. 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 lookahead in the IDE disk [22], 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 [22]; we suspect that block-based (rather than extent-based) allocation in ext3 accounts for Linux's slightly lower performance.

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.

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	—	—

Figure 13: Application-level benchmark results.

7.2 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; Figure 13 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 6.1.

8 RELATED WORK

HiStar was directly inspired by Asbestos, but differs in providing system-wide persistence, explicit resource allocation, and a lower-level kernel interface that closes known covert storage channels. While Asbestos 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 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 [1]. The ADEPT-50 dynamically adjusted labels (essentially taint tracking) using the High-Water-Mark security model back in the late 1960s [10]; the idea has often resurfaced, for instance in IX [13] and LOMAC [7]. HiStar and its predecessor Asbestos are novel in that they make operations such as category allocation and untainting available to application programmers, where previous OSes reserved this functionality for security administrators. Decentralized untainting 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 capability-based KeyKOS [2] and its successor EROS [23]. 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 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 [13], or virtualized [18] superuser privileges. Several operating systems including Linux support POSIX capabilities, which can permit some superuser privileges while disabling others.

Plan 9 [17] 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 [21]. However, HiStar's protection domains are not hierarchical like Multics rings. HiStar gates are more like doors in Spring [8].

Decentralized untainting, while new in operating systems, was previously provided by programming languages, notably Jif [14]. 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 storage, trusted input files, administration, etc., which avoids many issues HiStar needs to address.

Singularity [9] 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 [11] 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.

9 LIMITATIONS

We believe HiStar provides a good environment to develop secure applications with small trusted code size. Nonetheless, the system has 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 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. 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.

10 SUMMARY

HiStar is a new operating system that provides strict information flow control without superuser privilege. Narrow interfaces allow for a small trusted kernel of less than 16,000 lines, on which a Unix-like environment is implemented mostly as untrusted user-level library code. A new container abstraction lets administrators manage and revoke resources for processes they cannot observe. 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. Benchmarks show HiStar performs competitively with Linux and OpenBSD.

ACKNOWLEDGMENTS

We thank Hector Garcia-Molina, Michael Freedman, Ramesh Chandra, Constantine Sapuntzakis, Jim Chow, the anonymous reviewers, and our shepherd, Rob Pike, for their feedback. We also thank the Asbestos group, especially Steve VanDeBogart and Petros Efstathopoulos, who co-designed Asbestos labels. HiStar was in part inspired by some of the input Cliff Frey had to the Asbestos project. This work was funded by joint DARPA/NSF Cybertrust grant CNS-0430425.

REFERENCES

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

[2] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

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

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

[5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th SOSP*, pages 17–30, October 2005.

[6] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. Ottawa Linux Symposium, 2002.

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

[8] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. of the Summer 1993 USENIX*, pages 147–159, April 1993.

[9] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.

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

[11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the 2001 USENIX*, pages 29–40, June 2001. FREENIX track.

[12] LWIP. `http://savannah.nongnu.org/projects/lwip/`.

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

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

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

[16] OpenVPN. `http://openvpn.net/`.

[17] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3): 221–254, Summer 1995.

[18] H. Pötzl. *Linux-VServer Technology*, 2004. `http://linux-vserver.org/Linux-VServer-Paper`.

[19] K. 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/`.

[20] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. of the 13th SOSP*, pages 1–15, Oct. 1991.

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

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

[23] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th SOSP*, pages 170–185, December 1999.

[24] 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, 22–26 1996.

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