DOI:10.1145/2018396.2018419

Making Information Flow Explicit in HiStar

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

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 privacy-preserving, untrusted virus scanners and a dynamic Web server with only a few thousand lines of trusted code.

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 mind-set 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: they make it too difficult to predict the full implications of every action by untrusted code.[7] 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, let us consider anti-virus software, which often has full access to all files on a user's computer. There have been critical vulnerabilities discovered in virus scanners from Norton,[14] McAfee,[10] and others[15] that allow attackers to take full control of the scanner. Such vulnerabilities can easily be exploited to, at the very least, steal private data from millions of users. To prevent such a disaster, we might switch to the simpler, open-source ClamAV virus scanner. However, it has suffered from security vulnerabilities in the past,[21] and is over 40,000 lines of code—large enough that hand-auditing the system to eliminate vulnerabilities would be an expensive and lengthy process at best. 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 to keep the user's data private, thereby minimizing potential damage from ClamAV's vulnerabilities.

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, or corrupt those files. 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 trick another program, such as a mail server running on the same machine, into transmitting the data.
- The scanner can take over an existing process using debug mechanisms (e.g., *ptrace* on Unix), and send the data via 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 network packets.
- The scanner can use any number of less efficient and subtler techniques to impart the data to the update daemon—for example, use file locking to lock different ranges of the database, bind particular TCP or UDP port numbers, modulate memory or disk usage in a detectable way, or change the title of the scanner process.

This work was originally presented at the 7th Symposium on Operating Systems Design and Implementation and the 5th Symposium on Networked Systems Design and Implementation.

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.

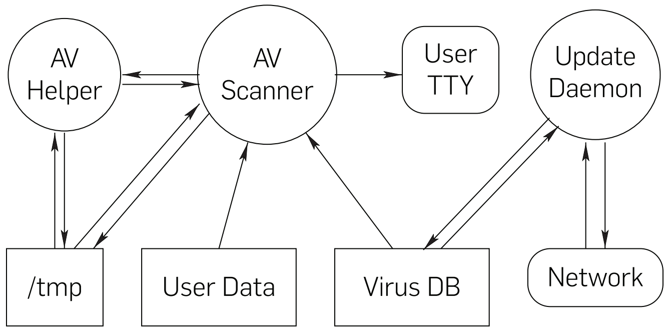
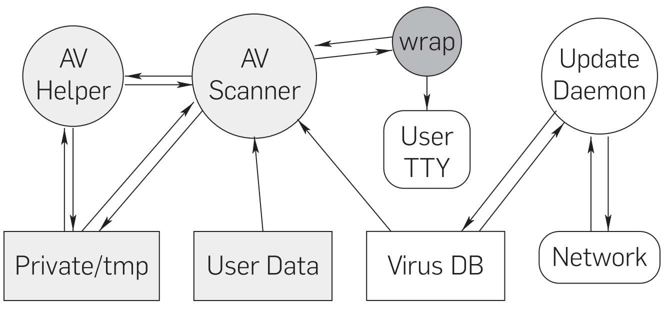


Figure 2. ClamAV running in HiStar. Lightly shaded components are *tainted*, which prevents them from conveying any information to untainted (unshaded) components. The strongly shaded `wrap` has untainting privileges, allowing it to relay the scanner's output to the terminal.



Some of these attacks can be mitigated by running the scanner with its own user ID in a *chroot* jail.[7] 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 files.[7] Other attacks, such as those involving sockets or System V IPC, can be prevented only 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. The second difference from Linux is that we have launched the scanner from a new, 110-line program called *wrap*, which has *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. As long as *wrap* is correctly implemented, ClamAV cannot leak the contents of the files it scans.

Although HiStar's tainting mechanism appears simple at a high level, making it work in practice requires addressing a number of challenges. First, there are myriad ways in which data can leak out onto the network, as illustrated above with Linux. How would an operating system like HiStar know to check the taint of the data being leaked for each and every one of them? Second, a typical OS kernel already provides a wide range of protection mechanisms, including user IDs, process memory protection, *chroot* jails, and so on. How can we avoid further complicating the kernel with yet another mechanism, or at very least, avoid unexpected interactions between the many disparate protection mechanisms? Finally, managing the tainting of files and the untainting privileges requires a separate mechanism, which can equally well be the target of attacks. One answer is to allow only the system administrator—root—access to this mechanism, but doing so both hampers the ability of other applications to use this mechanism and increases the amount of fully privileged code running as root.

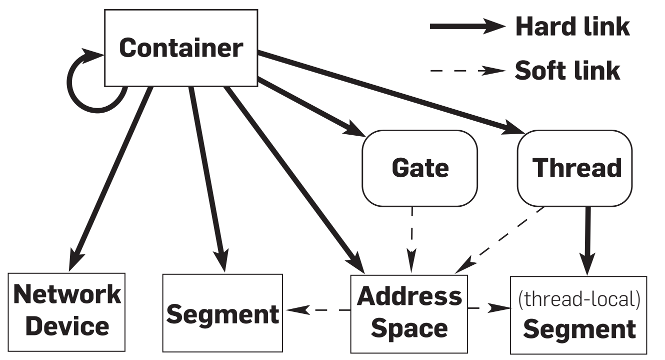
HiStar addresses these challenges with three key ideas. First, instead of implementing a traditional Unix interface, the kernel provides a lower-level interface, consisting of six types of kernel objects and a small number of operations that make any information flows between objects *explicit*. This provides a correspondingly small number of places where the kernel must perform data flow checks. Second, the *only* protection mechanism provided by the kernel is an information flow control mechanism, which generalizes the intuition behind taint. All other forms of protection, including Unix user IDs, process memory protection, and tainting itself, are implemented *in terms of* information flow control. This both reduces the amount of trusted kernel code and avoids any ambiguity about how the mechanisms will form a coherent policy. Finally, HiStar's information flow control mechanism is egalitarian, meaning that it can be used by any process, not just by superuser, which further reduces the amount of fully trusted code.

Though we used the virus scanner as an example, many security problems can be couched in terms of information flow. For example, protecting users' private profiles 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 files. Protecting passwords means ensuring that whatever code verifies them can reveal only the single bit signifying whether or not authentication succeeded. The rest of this paper describes how HiStar provides a new, Unix-like environment in which small amounts of code can secure much larger, untrusted applications by enforcing such policies.

2. DESIGN

The HiStar kernel is organized around six object types, shown in Figure 3: a *segment* (a variable-length byte array similar to a file), an *address space* (a mapping from virtual memory addresses to segment object names), a *network*

Figure 3. Kernel object types in HiStar. *Soft links* name objects by a particular ⟨container ID, object ID⟩ container entry. Threads and gates are represented by rounded rectangles to indicate they are the only objects that have ownership privileges.



device (which can send and receive packets), a *thread* (a set of CPU registers, along with the name of an address space object), a *gate* (an IPC mechanism), and a *container* (a directory-like object in which all other objects reside).

Each object has a unique 64-bit *object ID* and a *label* that is used to control information flow to or from that object. All of the state accessible to user processes is stored in kernel objects (except for a few global variables, such as the counter used to allocate fresh object IDs). Thus, to read or write any data, processes must invoke the kernel (e.g., issue a system call or trigger a page fault to access a memory-mapped file). Upon receiving such a request, the kernel compares the labels of the currently executing thread and the objects being accessed to decide whether the operation should be permitted. While it is not possible to interpose on every read and write to memory-mapped files, the kernel remembers all active memory mappings and invalidates them when it suspects access should no longer be allowed (e.g., when a thread's label changes).

2.1. Labels

Before discussing the kernel interface further, we first describe HiStar's labels more precisely. HiStar associates a label with every kernel object. The purpose of a label is to provide a conservative estimate of what kind of data might be present in an object.

Generalizing the virus-scanner example shown in Figure 2, there may be multiple kinds of secret data in a system, perhaps belonging to different users. HiStar uses the notion of secrecy *categories* to distinguish between different kinds of secret data (a category is just an opaque 64-bit identifier), and a label is simply a set of categories. For example, in Figure 2, lightly shaded components have one specific secrecy category in their label; processes and files not shown in the figure can be labeled with one or more other categories. Data can flow from object A to object B only if B's label includes all of the secrecy categories in A's label. This is similar to the Bell-LaPadula model,[1] and ensures that any data marked secret remains in objects marked secret.

Secrecy categories help control where secret data can end up, but it is also important to control where data comes from. For instance, the virus scanner may want to ensure its virus database has not been corrupted by another application, and one user may want to prevent other users from overwriting his files. To address this problem, HiStar provides a second type of category—an *integrity* category. The type of a category is stored in the high bit of the category's 64-bit identifier, but in the rest of this paper we will use the notation c_r to indicate a secrecy (read) category and c_w to indicate an integrity (write) category. Object labels can include both secrecy and integrity categories, but the rules for integrity categories are the opposite of secrecy: data can flow from A to B only if A's label includes all of the integrity categories in B's label. This is analogous to the Biba integrity model,[2] and ensures that high-integrity files can be modified only by high-integrity sources.

Given these two types of categories, we can formalize when data can flow between two objects. For any two objects A and B, with labels L_A and L_B, data can flow from A to B if and only if every secrecy category in L_A is present in L_B, and every integrity category in L_B is present in L_A. This relation is checked frequently by HiStar, and we denote it by $L_A \sqsubseteq L_B$ (pronounced L_A *can flow to* L_B). Note that the $\sqsubseteq$ relation is transitive, meaning that one can understand if data can flow between two objects without having to consider all possible intermediate objects through which the data may flow.

While these rules ensure that secret data can propagate only to secret-labeled objects, a practical system requires occasionally extracting secret data from the system. For example, the *wrap* program shown in Figure 2 needs to send the output of the virus scanner to the user's terminal. HiStar allows this using the notion of category *ownership*. Each thread T, in addition to having a label L_T, owns a set of categories O_T, and these categories are ignored when performing operations on behalf of T. For example, T can read object A if $L_A - O_T \sqsubseteq L_T - O_T$, which we write as $L_A \sqsubseteq_{O_T} L_T$ (pronounced L_A *can flow using privileges* O_T *to* L_T).

In our virus scanner example, a user's files could be labeled $L_f = \{u_r, u_w\}$, where u_r and u_w are categories owned by the user that protect the secrecy and integrity of that user's data. The virus scanner runs with label $L_s = \{u_r\}$ and empty ownership set $O_s = \varnothing$, which allows it to read the user's files ($L_f \sqsubseteq_{O_s} L_s$), but not to modify them ($L_s \not\sqsubseteq_{O_s} L_f$) or export them (since the network is labeled $\varnothing$ and $L_s \not\sqsubseteq_{O_s} \varnothing$). The *wrap* process has label $L_w = \varnothing$ and ownership set $O_w = \{u_r\}$, which allows it to read data from the scanner ($L_s \sqsubseteq_{O_w} L_w$) and write it to the user's terminal ($L_w \sqsubseteq_{O_w} \varnothing$). Typically, a process trusted by the user owns both u_r and u_w, giving it the privileges to read, write, and export that user's data.

A key property of HiStar's labels is that ownership of one category confers no privileges with respect to other categories. This means that, for any secrecy category c_r, data labeled with c_r will flow only to objects labeled with c_r, unless a thread that owns c_r intervenes, and vice versa for integrity. This makes it possible to provide end-to-end guarantees on how different components can affect each other, by just inspecting components that own the relevant categories. For example, in Figure 2, it suffices to examine *wrap* to understand how shaded components can affect unshaded ones.

2.2. Labeling kernel state

HiStar's kernel enforces information flow control by associating a label with every piece of user-visible state in the system—such as the registers of a thread, the length of a segment, and even the label of an object itself—and using the $\sqsubseteq$ relation to decide if a given thread should be allowed to observe or modify that state. As long as every piece of kernel state that can, directly or indirectly, influence execution of user code has a consistent label, then $\sqsubseteq$'s transitivity guarantees security: if data from thread A can affect some piece of kernel state X, and data from X can flow to some other thread B, then we must have checked that $L_A \sqsubseteq L_X$ and $L_X \sqsubseteq L_B$, which implies $L_A \sqsubseteq L_B$, so it was safe for data to flow from A to B. The key to ensuring transitivity lies in associating a consistent label with each piece of kernel state regardless of how the user code tries to, directly or indirectly, learn its value or modify it.

The bulk of HiStar's kernel state resides in kernel objects. For example, the simplest type of object is a segment, which contains a variable-length byte array. When thread T attempts to read segment S, either by issuing a system call or by triggering a page fault, the kernel checks that $L_S \sqsubseteq_{O_T} L_T$. Likewise, when thread T attempts to write S, the kernel checks that $L_T \sqsubseteq_{O_T} L_S \sqsubseteq_{O_T} L_T$. (The kernel ensures that data is allowed to flow from T to S and vice versa; our experience suggests it is difficult to write to an object without receiving some information as to whether the write succeeded.) As another example, a network device object's payload is (logically) all of the packets on the Ethernet network. To send or receive a packet, a thread must be able to write or read the network device, respectively, with rules identical to those for a segment.

Each object also contains the object's ID, the label, and a 64-byte mutable, user-defined metadata buffer (used by user-level code to, for instance, track modification times). The metadata buffer can logically be thought of as part of the mutable object contents, and is subject to the same read and write rules as the object contents. On the other hand, the label of an object O, L_O, presents a challenge: how should we label L_O's bytes? Suppose that we used L_O as the label of the entire object O, including L_O itself. If a thread tries to read O and is denied access, it learns something about the contents of L_O, even though this flow was prohibited.

To solve this chicken-and-egg problem, HiStar logically associates O's parent container's label with the bytes comprising L_O (as a special case, the root container is its own parent). Furthermore, because O might reside in multiple parent containers, HiStar requires that object labels be specified at creation and then immutable (except for threads, as we discuss later).

To deal with on-disk state, HiStar provides a single-level store: on bootup, the entire system (including threads) is restored from the most recent on-disk snapshot. This eliminates the need for trusted boot scripts to reinitialize processes that would not survive a reboot on traditional operating systems. It also achieves economy of mechanism by allowing the file system to be implemented with the same kernel abstractions as virtual memory, without any additional mechanisms for labeling on-disk state.

Finally, the kernel maintains a small amount of state outside of kernel objects, namely, the counter used to generate new object and category IDs. Newly allocated IDs must have two properties: first, they must be unique, and second, they must disclose almost no information about the state of the system, such as the number of previously allocated objects (*almost* because by definition, a new ID reveals the fact that this exact ID value was never allocated before). HiStar generates IDs by encrypting a counter with a block cipher. Since the block cipher is a pseudo-random one-way function, an attacker cannot learn any information from the value of the ID itself, and since the block cipher is a permutation, the IDs are unique.

2.3. Threads

Each thread T has a label L_T and an ownership set O_T, which can be changed through two mechanisms. First, a thread can allocate a fresh category by invoking the system call

- cat_t *create_category* (cat_type t),

which chooses a previously unused category, c, and adds c to O_T. The type of the category (secrecy or integrity) is specified by t. At this point, T is the only thread that owns c, and since c was never used before, granting T ownership of c confers no other privileges. In this sense, labels are egalitarian: no thread has any inherent privileges with respect to categories created by other threads. T can also drop categories from its ownership set.

T may change its own label through the system call

- int *self_set_label* (label_t L),

which sets $L_T \leftarrow L$, as long as $L_T \sqsubseteq_{O_T} L$. This can, for example, let T read a tainted object, or to untaint its label in categories it owns. HiStar also includes a clearance mechanism,[18] which prevents a thread from arbitrarily raising its label to read all possible data, but its discussion is omitted here for clarity.

A thread T can allocate new objects with label L as long as $L_T \sqsubseteq_{O_T} L$. Threads and gates (which will be discussed shortly) can be created with an ownership set O as long as $O \subseteq O_T$.

2.4. 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. To ensure that such resources can nonetheless be reclaimed, HiStar provides hierarchical control over object allocation and deallocation through *containers*. 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 3 shows the possible links between containers and other types of objects.

When allocating an object, a thread must specify the container into which to place the object. For example, to create a container, thread T makes the system call

- id_t *container_create* (id_t C, label_t L).

Here C is the object ID of an existing container, into which the newly created container will be placed. L is the desired label for the new container. The system call succeeds only if T can write to C (i.e., $L_T \sqsubseteq_{O_T} L_C \sqsubseteq_{O_T} L_T$) and allocate an object of label L

(i.e., $L_T \sqsubseteq_{O_T} L$). Objects can be likewise *unreferenced* from container C by any thread that can write to C. 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.

Containers also help HiStar address a possible covert channel through object reference counting. Any thread T can create a hard link to segment S in container C if it can write C (i.e., $L_T \sqsubseteq_{O_T} L_C \sqsubseteq_{O_T} L_T$). T can thus prolong S's life even without permission to modify S; in our virus-scanner example from Figure 2, this might be the malicious scanner process signaling secret information by prolonging or not prolonging the life of the database file. Another thread T', such as the update process, could then remove any known links to S and observe whether it can still access S by its object ID, even if T was not allowed to communicate to T'.

To avoid this problem, most system calls name objects by ⟨container ID, object ID⟩ pairs, called *container entries*. For T' to use container entry $\langle C, S\rangle$, C must contain a link to S and T' must be able to read C (i.e., $L_C \sqsubseteq_{O_{T'}} L_{T'}$). In the virus-scanner example, the untainted update process would not be able to use any container entry created by the tainted scanner. Container entries allow the kernel to check if a thread has permission to know if the object exists, in addition to any other label checks necessary to access the object.

2.5. Address spaces

Every running thread has an associated address space object containing a list of VA → ⟨S, *offset*, *npages*, *flags*⟩ mappings. VA is a page-aligned virtual address. $S = \langle C, O\rangle$ 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 $L_T \sqsubseteq_{O_T} L_A \sqsubseteq_{O_T} L_T$, and can observe or use A only if $L_A \sqsubseteq_{O_T} L_T$. When launching a new thread, one must specify its address space and program counter. The system call *self_set_as* 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 $S = \langle C, O\rangle$ and *flags*. If *flags* allows the access mode, the kernel checks that T can read C and O ($L_C \sqsubseteq_{O_T} L_T$ and $L_O \sqsubseteq_{O_T} L_T$). If *flags* includes writing, the kernel additionally checks that T can modify O ($L_T \sqsubseteq_{O_T} L_O$). 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.

2.6. Gates

Gates provide protected control transfer, allowing a thread to jump to a predefined program counter in another address space with additional privilege. A gate object G has an ownership set O_G, a guard set G_G, and thread state, including the container entry of an address space, an initial program counter and stack pointer, and some closure arguments for the initial function. The guard set controls what other threads can invoke this gate, by requiring the caller to own all categories in G_G. A thread T can allocate a gate G only if $O_G \subseteq O_T$. A thread T' invoking G must specify a requested ownership set, O_R, to acquire upon invocation; invocation is permitted when $O_T \subseteq G_G$ and $O_R \subseteq (O_T \cup O_G)$. Gate objects are largely immutable (and thus subject to the parent container's label); the gate label L_G applies only to the gate object's (rarely-used) metadata.

Gates are often 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. Gates can also be used to transfer privilege. The use of gates is discussed further in Section 3.5.

3. UNIX LIBRARY

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.

HiStar's Unix environment is implemented in a library 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. All of these abstractions are provided at user level, without any special privilege from the kernel. Thus, all information flow, such as obtaining 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, compromises only threads that trigger the bug—an attacker can exercise only 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.

Most GNU software runs on HiStar without any source code modifications, including bash, gcc, gdb, and X; the main exception is OpenSSH, which requires small changes for user authentication and login code. The rest of this section discusses the design and implementation of our Unix emulation library.

3.1. 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 for invalid read or write requests. A file's length is defined to be the segment's length. 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. A mutex in the directory segment serializes operations; 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.

Since file system objects correspond to HiStar kernel objects, permissions are specified in terms of labels, and are enforced by the kernel, not by the untrusted library file system code. For example, a file that should be accessible by only one user would be labeled $\{u_r, u_w\}$, where only that user owns u_r and u_w. A world-readable file that can be modified by only that user would be labeled $\{u_w\}$. 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.

3.2. Processes

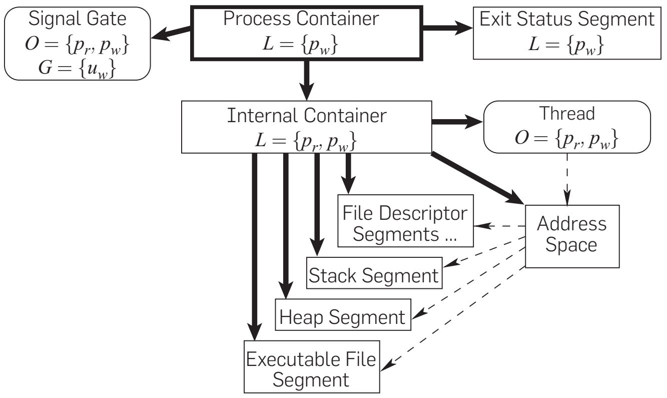
A process in HiStar is a user-space convention. Figure 4 illustrates the kernel objects that make up a typical process. Each process P has two categories, p_r and p_w, that protect its secrecy and integrity, respectively. Threads in a process typically own $\{p_r, p_w\}$, granting them full access to 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, such as a gate to receive signals from other processes (described in detail in the OSDI paper[18]) and a segment to store the process's exit status. The process container and exit status segment are labeled $\{p_w\}$, allowing other processes to read them, but not modify them (since other processes do not own this process's p_w). The internal container, address space, and segment objects are labeled $\{p_r, p_w\}$, preventing direct access by other processes.

3.3. File descriptors

All of the state typically associated with a file descriptor, such as the current seek position and open flags, is stored in a *file descriptor segment* in HiStar. 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 $\{fd_r, fd_w\}$, where categories fd_r and fd_w grant read and write access to the file descriptor state. Access to the file descriptor is granted by granting ownership of $\{fd_r, fd_w\}$. 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, a shared descriptor segment is deallocated only when it has been closed and unlinked from the container of each process.

Figure 4. Structure of a HiStar process. A process container is represented by a thick border. Not shown are some label components that, e.g., ensure other users cannot read the exit status of this process. Bold and dashed lines represent hard and soft links.



3.4. 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 own $\{u_r, u_w\}$, and a user's private files would have a label of $\{u_r, u_w\}$. One consequence of this design is that a single process can possess the privilege of multiple users, or perhaps multiple user roles, something that is hard to implement in Unix. On the other hand, our prototype does not support access control lists. (One way to implement access control lists would be to allocate a pair of categories for each ACL and to create a gate that would invoke code to evaluate the ACL rules and selectively grant ownership of these categories.) The authentication service, which verifies user passwords and grants user privileges, is described in more detail in Zeldovich et al.[18]

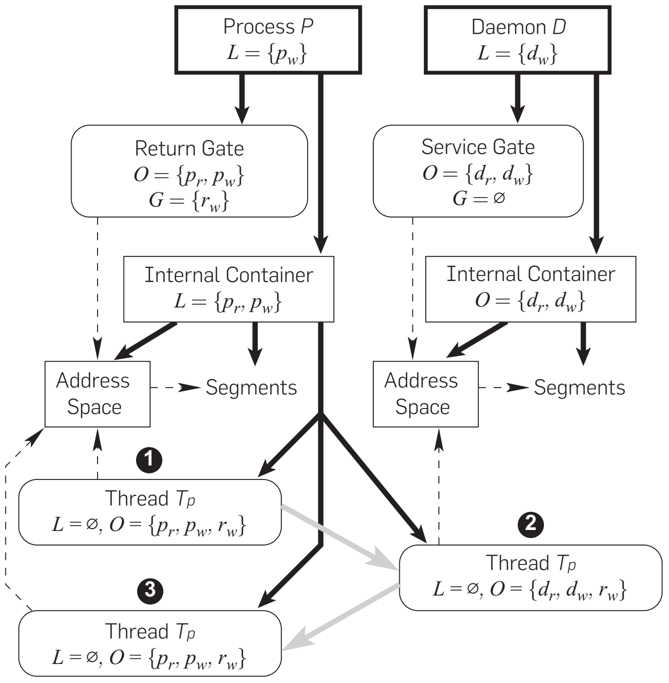
3.5. Gate calls

Gates provide a mechanism for implementing IPC. As an example, consider a phonebook service that allows looking up people's phone numbers by their name. Storing all names in a file may be undesirable, since users could easily obtain a list of all names. A HiStar process could provide this service by creating a *service gate* whose initial program counter corresponded to a function that looks up a name in a database that is accessible only to that process.

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_w is allocated to prevent arbitrary threads from invoking the return gate; the return gate's guard set requires ownership of r_w to invoke the gate, and the caller grants ownership of r_w when invoking the service gate. Figure 5 illustrates a gate call from process P to daemon D.

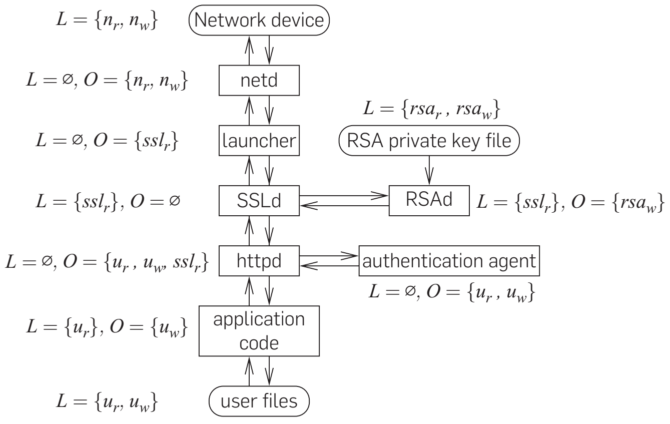
While this design prevents a user from enumerating the daemon's private database, it does not ensure privacy of users' queries. If users do not trust the daemon to keep their queries private, they can enforce the privacy of their queries, as follows. The calling thread allocates a new secrecy category x_r and invokes the service gate in step 2 with a label of $\{x_r\}$ (instead of $\varnothing$). This thread can now read D's address space and any of D's segments, by virtue

Figure 5. Objects involved in a gate call operation. Thick borders represent process containers, bold lines represent hard links, and dashed lines represent soft links. r_w 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.



Process P $L = \{p_w\}$
Daemon D $L = \{d_w\}$
Return Gate $O = \{p_r, p_w\}$ $G = \{r_w\}$
Service Gate $O = \{d_r, d_w\}$ $G = \varnothing$
Internal Container $L = \{p_r, p_w\}$
Internal Container $O = \{d_r, d_w\}$
Address Space
Segments
Address Space
Segments
1
Thread T_p $L = \varnothing, O = \{p_r, p_w, r_w\}$
2
Thread T_p $L = \varnothing, O = \{d_r, d_w, r_w\}$
3
Thread T_p $L = \varnothing, O = \{p_r, p_w, r_w\}$

Figure 6. Architecture of the HiStar Web server. Rectangles represent processes. Rounded boxes represent devices and files. Arrows indicate communication, including gate calls. Not shown are process read and write categories from Figure 4.



$L = \{n_r, n_w\}$
Network device
$L = \varnothing, O = \{n_r, n_w\}$
netd
$L = \{rsa_r, rsa_w\}$
$L = \varnothing, O = \{ssl_r\}$
launcher
RSA private key file
$L = \{ssl_r\}, O = \varnothing$
SSLd
RSAd
$L = \{ssl_r\}, O = \{rsa_w\}$
$L = \varnothing, O = \{u_r, u_w, ssl_r\}$
httpd
authentication agent
$L = \varnothing, O = \{u_r, u_w\}$
$L = \{u_r\}, O = \{u_w\}$
application code
$L = \{u_r, u_w\}$
user files

of owning d_r, but not modify them, since that would violate information flow constraints in category x_r. To continue executing, the thread makes a writable copy of the address space and its segments, labeled $\{d_r, d_w, x_r\}$. This effectively *forks D* to create a tainted clone. The thread can now read the database and return data to the caller, but cannot divulge any data to anyone that does not own x_r. Two considerations arise in this case. First, the tainted copy must be stored in some container. Since the thread is labeled x_r, the kernel does not allow it to store objects in *D*'s container (otherwise, the thread could leak information about the caller's private data to *D*) . Thus, before invoking the gate, the caller creates a container labeled x_r that can be used after invoking the gate. Second, when the thread returns to *P*'s address space through the return gate, its label still contains x_r. To remove x_r from the thread's label, *P* must have added x_r to the return gate's ownership label prior to invoking *D*.

Forking on tainted gate invocation is not appropriate for every service. Stateless services, such as a database lookup, are usually well-suited to forking, whereas services with mutable shared state may want to avoid forking by refusing tainted gate calls.

4. HiSTAR WEB SERVER

To illustrate how HiStar's protection mechanisms can be used by a real application, this section describes the HiStar SSL Web server. Figure 6 shows the server's overall architecture. The Web server is built from mutually distrustful components to reduce the effects of a compromise of any single component.

The TCP/IP stack in HiStar is implemented by a user-space process called *netd*, which has access to the kernel network device.[18] *netd* provides a traditional sockets interface to applications. Incoming TCP connections from Web browsers are initially accepted by the *launcher*. For each connection, the *launcher* spawns *SSLd*, to handle the SSL connection with the user's Web browser, and *httpd*, to process the user's plaintext HTTP request. The *launcher* then relays data between *SSLd* and the TCP connection. *SSLd*, in turn, uses the *RSAd* daemon to establish an SSL session key with the user's Web browser by generating an RSA signature using the SSL certificate private key kept by *RSAd*.

httpd receives the user's decrypted HTTP request from *SSLd* and extracts the user's password and the requested URL. It then authenticates the user by sending the user's password to that user's *password checking agent* from the HiStar authentication service.[18] If the authentication succeeds, *httpd* receives ownership of the user's secrecy and integrity categories, u_r and u_w, and executes the *application code* with the user's privileges (e.g., generating a PDF document). Application output is sent by *httpd* to the user's Web browser via *SSLd* for encryption.

4.1. 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. Figure 7 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

Figure 7. Components of the HiStar Web server, their complexity measured in lines of C code (not including libraries such as libc), their label and ownership, and the worst-case results of the component being compromised. The netd TCP/IP stack is a modified Linux kernel; HiStar also supports the lwIP TCP/IP stack, consisting of 35,000 lines of code, which has lower performance.

Component	Lines of Code	Label	Ownership	Effects of Compromise
netd	350,000	$\varnothing$	$\{n_r, n_w\}$	Equivalent to an active network attacker; subject to same kernel label checks as any other process
launcher	310	$\varnothing$	$\{ssl_r\}$	Obtain plaintext requests, including passwords, and subsequently corrupt user data
SSLd	340,000	$\{ssl_r\}$	$\varnothing$	Corrupt request or response, or send unencrypted data to same user's browser
RSAd	4,600	$\{ssl_r\}$	$\{rsa_r\}$	Disclose the server's SSL certificate private key
httpd	300	$\varnothing$	$\{u_r, u_w, ssl_r\}$	Full access to data in attacker's account, but not to other users' data
authentication	320	$\varnothing$	$\{u_r, u_w\}$	Full access to data of the user whose agent is compromised, but no password disclosure
application	680,000+	$\{u_r\}$	$\{u_w\}$	Send garbage (only to the same user's browser), corrupt user data (for write requests)

they are malicious. The application code is confined by the user's secrecy category, u_r, and it is *httpd*'s job to ensure that the application code is labeled with u_r when *httpd* runs it. Although the application code owns the user's integrity category, u_w, this gives it the privilege to write only that one user's files, and not to export them. Ownership of u_w is necessary to allow the application code to read data not labeled with u_w. If the application code were to be labeled with u_w, it would be restricted to reading only data labeled u_w, which would exclude executables, shared libraries, and configuration files.

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

SSLd is not trusted to handle the SSL certificate private key. Instead, a separate and much smaller daemon, *RSAd*, has access to the private key and provides an interface to generate RSA signatures for SSL session key establishment. Not shown in Figure 7 is a category owned by *SSLd* that ensures no other process can invoke *RSAd*. Since *SSLd* is confined by ssl_r, the kernel ensures that *SSLd* cannot indirectly divulge any data to another process via its calls to *RSAd*, much as described in Section 3.5.

The HiStar authentication service used by *httpd* to authenticate users is described in detail in Zeldovich et al.,[18] but briefly, no code executes with all users' privileges, and the supplied password cannot be leaked even if the password checker is malicious. Our Web server does not use SSL client certificates for authentication. Doing so would require either trusting all of *SSLd* to authenticate users, or moving the client certificate code into the authentication agent. In comparison, the password checking agent is 320 lines of code.

One caveat of our prototype is its lack 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 an SSL session cache that runs in a separate persistent process and owns all ssl_r categories, at the cost of increasing the amount of trusted code.

5. RELATED WORK

HiStar was directly inspired by Asbestos,[4] but differs in providing systemwide persistence 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 optimizes comparisons between enormous labels, which so far we have not done in HiStar.

Flume[8] showed how to provide information flow control on top of the Linux kernel, and introduced a cleaner label system (which HiStar and this paper have adopted). Flume also proposed *endpoints* to help programmers reason about labels on standard Unix abstractions; adopting endpoints in HiStar's Unix library would similarly help HiStar programmers. DStar[19] extended information flow control to decentralized systems and developed the HiStar Web server. Loki[20] showed how hardware can partially enforce HiStar's labels, to reduce the amount of fully trusted kernel code.

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[9]; the idea has often resurfaced, for instance in IX[12] and LOMAC.[5] 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, which were not targeted by previous MAC systems.

Like HiStar, capability-based KeyKOS[3] and EROS[17] use a small number of kernel object types and a single-level store. HiStar's containers are 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.

The idea of using gates for protected control transfer dates back to Multics.[16] Unlike Multics rings, HiStar's protection domains are not hierarchical. HiStar gates are more like doors in Spring.[6]

Decentralized untainting, while new in operating systems, was previously provided by programming languages, notably Jif.[13] Jif can track information flow at the level of individual variables and perform most label checks at compile time. However, Jif relies on the operating system for storage, trusted input files, administration, etc., which avoids many issues HiStar needs to address.

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. Retrofitting MAC to a large existing kernel such as Linux can be error-prone, especially 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.

6. DISCUSSION AND LIMITATIONS

The current prototype of HiStar supports x86-64, i386, SPARC, and ARM computers. The fully trusted kernel, including device drivers for any given machine, is approximately 20,000 lines of code. We expect that drivers can eventually be moved to untrusted user-space processes with the help of IOMMU hardware. We have found performance to be reasonable for Unix applications.[18]

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 keep file access times; although possible to implement for some cases, 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).

While trusted components can control how secret data is revealed, it is difficult to reason about *what* secret data is revealed. For example, *wrap* can ensure the scanner's output is sent only to the user's terminal, but it would be difficult to safely reveal even one bit of information from the scanner's output to the public (e.g., are any of the user's files infected?), since we must conservatively assume that the scanner's output may reveal *any* bit about the user's data.

7. 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 20,000 lines, on which a Unix-like environment is implemented in 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. HiStar is available at http://www.scs.stanford.edu/histar/.

Acknowledgments

We thank Martin Abadi, Michael Reiter, and Michael Walfish for helping improve this paper, and many others that provided feedback on earlier papers.[18–20] This work was funded by joint NSF Cybertrust/DARPA grant CNS-0430425, by NSF Cybertrust award CNS-0716806, by the DARPA Application Communities (AC) program as part of the VERNIER project at Stanford and SRI International, and by a gift from Lightspeed Venture Partners. C

References

1. Bell, D.E., La Padula, L. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, March 1976.
2. Biba, K.J. *Integrity Considerations for Secure Computer Systems*. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, April 1977.
3. Bomberger, A.C., Frantz, A.P., Frantz, W.S., Hardy, A.C., Hardy, N., Landau, C.R., Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, 95–112.
4. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th SOSP* (Brighton, U.K., October 2005), 17–30.
5. Fraser, T. LOMAC: low water-mark integrity protection for COTS environments. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, May 2000), 230–245.
6. Hamilton, G., Kougiouris, P. The Spring nucleus: a microkernel for objects. In *Proceedings of the Summer 1993 USENIX* (Cincinnati, OH, April 1993), 147–159.
7. Krohn, M., Efstathopoulos, P., Frey, C., Kaashoek, F., Kohler, E., Mazières, D., Morris, R., Osborne, M., VanDeBogart, S., Ziegler, D. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems* (Santa Fe, NM, June 2005).
8. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R. Information flow control for standard OS abstractions. In *Proceedings of the 21st SOSP* (Stevenson, WA, October 2007), 321–334.
9. Landwehr, C.E. Formal models for computer security. *Comput. Surv. 13*, 3 (September 1981), 247–278.
10. Leyden, J. Anti-virus vulnerabilities strike again. *The Register*, March 2005. http://www.theregister. co.uk/2005/03/18/mcafee_vuln/
11. Loscocco, P., Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX* (Boston, MA, June 2001), 29–40, FREENIX track.
12. McIlroy, M.D., Reeds, J.A. Multilevel security in the UNIX tradition. *Softw. Pract. Exp. 22*, 8 (1992), 673–694.
13. Myers, A.C., Liskov, B. Protecting privacy using the decentralized label model. *Trans. Comput. Syst. 9*, 4 (October 2000), 410–442.
14. Naraine, R. Symantec antivirus worm hole puts millions at risk. *eWeek.com*, May 2006. http://www.eweek. com/article2/0,1895,1967941,00.asp
15. Peterson, D. Anti-virus rife with vulnerabilities. *digitalbond.com*, January 2008. http://www.digitalbond.com/index.php/2008/01/07/anti-virus-rife-with-vulnerabilities/
16. Schroeder, M.D., Saltzer, J.H. A hardware architecture for implementing protection rings. In *Proceedings of the 3rd SOSP* (New York, March 1972), 42–54.
17. Shapiro, J.S., Smith, J.M., Farber, D.J. EROS: a fast capability system. In *Proceedings of the 17th SOSP* (Island Resort, SC, December 1999), 170–185.
18. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D. Making information flow explicit in HiStar. In *Proceedings of the 7th OSDI* (Seattle, WA, November 2006), 263–278.
19. Zeldovich, N., Boyd-Wickizer, S., Mazières, D. Securing distributed systems with information flow control. In *Proceedings of the 5th NSDI* (San Francisco, CA, April 2008), 293–308.
20. Zeldovich, N., Kannan, H., Dalton, M., Kozyrakis, C. Hardware enforcement of application security policies. In *Proceedings of the 8th OSDI* (San Diego, CA, December 2008), 225–240.
21. Zoller, T. Clamav 0.94 and below—evasion and bypass due to malformed archive. April 2009. http://blog.zoller.lu/2009/04/clamav-094-and-below-evasion-and-bypass.html

Nickolai Zeldovich, Massachusetts Institute of Technology, CSAIL, Cambridge, MA.

Silas Boyd-Wickizer, Massachusetts Institute of Technology, CSAIL, Cambridge, MA.

Eddie Kohler, University of California, Los Angeles, CA.

David Mazières, Stanford University, Stanford, CA.