Hi-Fi: Collecting High-Fidelity Whole-System Provenance

Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel
Pennsylvania State University
University Park, PA
{djpohly,smclaugh,mcdaniel}@cse.psu.edu

Kevin Butler
University of Oregon
Eugene, OR
butler@cs.uoregon.edu

ABSTRACT

Data provenance—a record of the origin and evolution of data in a system—is a useful tool for forensic analysis. However, existing provenance collection mechanisms fail to achieve sufficient breadth or fidelity to provide a holistic view of a system's operation over time. We present Hi-Fi, a kernel-level provenance system which leverages the Linux Security Modules framework to collect high-fidelity whole-system provenance. We demonstrate that Hi-Fi is able to record a variety of malicious behavior within a compromised system. In addition, our benchmarks show the collection overhead from Hi-Fi to be less than 1% for most system calls and 3% in a representative workload, while simultaneously generating a system measurement that fully reflects system evolution. In this way, we show that we can collect broad, high-fidelity provenance data which is capable of supporting detailed forensic analysis.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security, Design

Keywords

data provenance, forensics, malware, reference monitor

1. INTRODUCTION

Data provenance, which is a detailed record of the origin and evolution of data in a system, is a useful tool in systems security. In its raw form, provenance data is simply a series of system events, such as a file being written or a process being created. Taken together, these events form the provenance record for that system, and examining this record can reveal detailed information about the system's secure or insecure operation. Previous works on data provenance have pointed out many such possibilities, such as performing intrusion detection [**?**] or identifying data which may have been exfiltrated from the system [**?**].

Provenance records are well suited to system forensics. Current forensic analysis techniques exploit the flexibility of event-based logs for a number of purposes. For example, audit logs can be used to evaluate ongoing compliance with real-world policies [**?**] or to create detailed reconstructions of several aspects of system state [**?**]. A complete provenance record provides an even richer set of information for this purpose (see Section **??**).

However, for a data provenance system to provide the holistic view of system operation required for such forensic applications, it must be complete and faithful to actual events. This property, which we call "fidelity," is necessary for drawing valid conclusions about system security. A missing entry in the provenance record could sever an important information flow, while a spurious entry could falsely implicate an innocuous process. As we discuss in Section **??**, these requirements can be achieved by designing the provenance collection mechanism around the reference monitor concept [**?**]. In particular, this mechanism must provide complete mediation for events which should appear in the record.

The following scenario illustrates this need: Alice runs a high-profile website. One day, her web server is infected by the (hypothetical) PwnHP worm. PwnHP takes control of a website's behavior by infecting the system's PHP binary. It also starts a daemonized process which periodically connects to a command and control server for instructions. These connections alert Alice to the fact that something is amiss.

Fortunately, Alice is collecting provenance data for this system. She retrieves the logs from her append-only storage server and begins to investigate. First, she locates one of the outgoing connections in the provenance record and traces the process provenance back to the original compromised thread in her web server. She can then follow the provenance trail forward to see the modified PHP binary, as well as all of the malicious behavior that it performs when executed. Alice can then proceed with confidence in restoring her system to a good state.

One lesson we can learn from this story is that forensic investigation requires a definition of provenance which is broader than just file metadata. What is needed is a record of *whole-system provenance* which retains actions of processes, IPC mechanisms, and even the kernel. These "transient" system objects can be meaningful even without being an ancestor of any "persistent" object. The command-and-control daemon on Alice's server, for example, was significant because it was a *descendant* of the compromised process. If the provenance system had deemed it unworthy of inclusion in the record, she could not have traced the outgoing connections to the compromise.

In this paper, we present Hi-Fi, a provenance system designed to collect high-fidelity whole-system provenance. Hi-Fi is the first provenance system which can collect a *complete* provenance record from early kernel initialization through system shutdown. Unlike existing provenance systems, it accounts for all kernel actions as well as application actions. Hi-Fi can also collect *socket provenance*, creating a system-level provenance record that spans multiple hosts. Furthermore, it solves a number of design and implementation problems unique to this work.

We evaluate Hi-Fi in two ways. First, we demonstrate its ability to capture behavior on a system running malicious software. We create a tool which performs common malicious actions such as creating a backdoor account, establishing persistence in the system, and exfiltrating sensitive data. In each case, inspection of the system provenance record revealed the malicious actions. Second, we evaluate Hi-Fi's performance for individual system calls and for a system-call heavy workload. We observe an overhead of less than 1% for most system calls, and a maximum of 6% for the `read` system call. For an I/O-bound workload, the average overhead is less than 3%.

2. BACKGROUND

Maintaining provenance records is a well-established practice in fields which deal with physical artifacts, but provenance for digital artifacts is a comparatively new application. The earliest implementations of digital provenance focused on highly structured, special-purpose data. One such system is Trio [?], a database management system that stores the provenance of its records. Many other special-purpose systems exist, such as Panda [?], which focuses on specific workflows, and provenance aware Condor [?], which collects provenance for jobs on a specific batch system.

To support forensic analysis, however, we need the ability to trace arbitrary, unstructured data. This requires general-purpose, system-level provenance collection. The first such provenance system, Lineage File System [?], accomplished this by intercepting system calls in a modified Linux kernel. When an application executed one of these calls, a record describing the action would be written to the `printk` buffer and stored in a MySQL database. The same system-call approach is used by more recent systems, such as PASSv2 [?], which handles ten different system calls, and Forensix [?], which intercepts around seventy-five. These systems analyze the arguments to system calls and write provenance data to log files on the disk. Unfortunately, system-call interception cannot produce a complete provenance record, because the kernel itself does not use system calls. Kernel-initiated actions, such as executing the interpreter for Alice's PHP scripts, are therefore not captured at the system-call layer.

Another option for collecting system-level provenance is to instrument the filesystem layer (e.g., the Linux VFS). This is the approach taken by the Story Book provenance system [?]. Story Book is designed as a framework which allows multiple "provenance sources" to collect data. One of the provided sources is a filesystem implemented using the FUSE API [?]. This filesystem acts as a layer between the kernel and an existing filesystem, capturing file activity and writing the resultant provenance data to a custom transactional storage system. Both the kernel and applications access files through the VFS, so this approach can generate a complete record of file activity. However, this does not provide the *whole-system* view of provenance needed for forensics.

2.1 Linux Security Modules

In order to create a system which does provide the needed properties, we draw on the three design goals of the reference monitor concept [?]. Tamperproofness, which states that a system cannot be made to behave incorrectly, will ensure that our provenance collector does not generate spurious or inaccurate records. Complete mediation guarantees that every access is handled, whether initiated by an application or by the kernel. For a provenance collector, this ensures that every legitimate event will appear in the record. Finally, if the collector is simple enough to be verified, then we can be certain that the first two properties hold. Taken together, these three conditions guarantee fidelity of the provenance record.

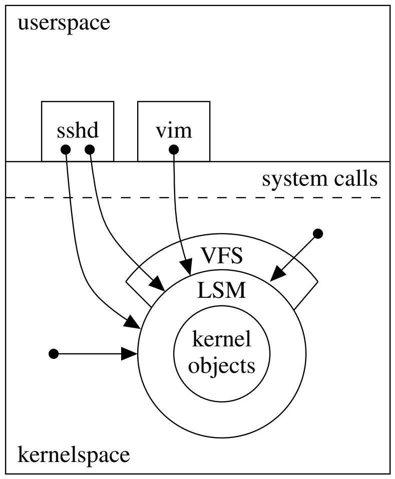


Figure 1: Complete mediation with LSM

Current approaches to provenance systems do not provide sufficient fidelity. Our system overcomes this by building on a framework intended for complete mediation. Linux Security Modules, or LSM, is a framework which was originally designed for integrating custom access control mechanisms into the Linux kernel [?]. It does this by mediating access, not to system calls, but to kernel objects themselves, as Figure **??** illustrates. The LSM framework comprises a set of hooks which are carefully placed throughout the kernel. Security modules can provide an implementation for any of these hooks, which are executed just before the corresponding access takes place. The placement of these hooks has been repeatedly analyzed and refined [?, ?, ?, ?] to ensure that every access is mediated.

The designers of the LSM framework are deliberate in establishing where this mediation takes place. In particular, they identify several issues with system-call interception: that it "is not race-free, may require code duplication, and may not adequately express the full context needed to make security policy decisions" [?]. LSM was created to avoid these problems and provide complete mediation, which is required for high-fidelity provenance collection.

3. DESIGN

Hi-Fi consists of three components: the provenance collector, the provenance log, and the provenance handler. Figure **??** depicts the interaction between these components. The collector is an LSM; as such, it resides in kernelspace and is notified whenever a kernel object access is about to take place. When invoked, the collector constructs an entry describing the action and writes it to the provenance log. The log is a buffer which presents these entries to userspace as a file. The provenance handler can then access this file using the standard file API, process it, and store the provenance record. The handler used in our experiments simply copies the log data to a file on disk, but it is possible to implement a custom han-

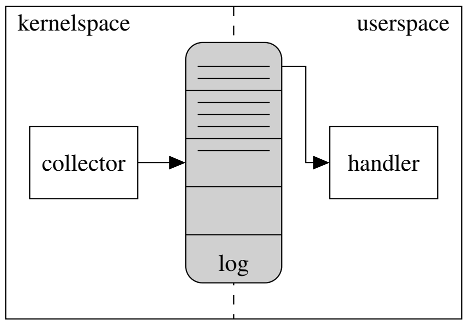


Figure 2: Architecture of Hi-Fi

dler for any purpose, such as post-processing, graphical analysis, or storage on a remote host.

3.1 Threat Model

We define a threat against our system as any way of compromising the fidelity of the provenance record during collection. Hi-Fi maintains the fidelity of provenance collection under any userspace compromise. This is a strictly stronger guarantee than those provided by current system-level provenance collection systems. In the event of a kernel-level compromise, the adversary will be able to tamper with the compenents of the provenance collector. However, the integrity of data *up to and including* the kernel compromise can be protected by an isolated disk-level versioning system [**?**] or a strong write-once read-many storage system [**?**]. In fact, since provenance data never changes after being written, a storage system with strong WORM guarantees is particularly well-suited to this task. For socket provenance, Hi-Fi guarantees that incoming data will be recorded accurately; to prevent on-the-wire tampering by an adversary, standard end-to-end protection such as IPsec should be used.

3.2 Provenance Collector

The main component of Hi-Fi is the in-kernel provenance collector, which is responsible for observing provenance-generating events. The collector consists of a number of LSM hooks which mediate operations on kernel objects. Table **??** lists all of the hooks which generate provenance data; several other hooks are used for internal memory management. For each hook, the collector gathers the relevant context from the kernel and writes one or more entries to the provenance log. By mediating the appropriate kernel objects, we are able to capture a wide variety of events:

- Reads and writes to file descriptors, including regular files, device files, and pipes.
- File operations: renaming, changing permissions, etc.
- Inter-process communication, such as shared memory, message queues, and UNIX domain sockets.
- Network communication between provenanced hosts.
- Program execution with full arguments and environment.
- Creation and deletion of credential objects (creds), which represent both process and kernel actions.
- User transitions, e.g., `login` changing to the authenticated user and group, `passwd` escalating to root by setuid execution, or `sshd` dropping privileges.

Kernel object	LSM hook
Inode	`inode_init_security`
	`inode_free_security`
	`inode_link`
	`inode_unlink`
	`inode_rename`
	`inode_setattr`
	`inode_readlink`
	`inode_permission`
Open file	`file_mmap`
	`file_permission`
Program	`bprm_check_security`
	`bprm_committing_creds`
Credential	`cred_prepare`
	`cred_free`
	`cred_transfer`
	`task_fix_setuid`
Socket	`socket_sendmsg`
	`socket_post_recvmsg`
	`socket_sock_rcv_skb`
	`socket_dgram_append`
	`socket_dgram_post_recv`
	`unix_may_send`
Message queue	`msg_queue_msgsnd`
	`msg_queue_msgrcv`
Shared memory	`shm_shmat`

Table 1: LSM hooks used to collect provenance

These events provide a comprehensive view of a system's history, including the entire process execution tree, the complete filesystem structure, and explicit information flows that may include network communication. These features can also be reconstructed for any given point in the past.

3.3 Provenance Handler

The responsibility of the provenance handler is to interpret, process, and store the provenance data after it is collected, and it should be flexible enough to support different needs. Consider the following examples. Alice, the website administrator we met earlier, has a dedicated provenance storage server with a huge disk. She does not want to do any extra processing or storage on her already overloaded web server; she just wants to move the provenance data over the LAN to her storage server as quickly as possible. Bob, on the other hand, is a provenance-curious researcher who would like to gather data from a number of volunteers. He would like the data formatted according to the Open Provenance Model [**?**] and uploaded to his web server in XML format. Alice and Bob have very different processing and storage needs for their provenance data. With an existing provenance system, their data would be stored in a database on disk before they could choose how to handle it.

Hi-Fi does not impose such limitations. Instead, we decouple the provenance handler from the collection process, allowing the system administrator to implement the handler according to the needs of the system. In our example, Alice can create a simple Bash script which pipes provenance data through `ssh` directly to her storage server. Bob is free to create a more complex handler which reads the log, uses a Java library from the OPM website to build the model and convert it to XML, and executes an HTTPS request to submit the document to his online database. He can then distribute this program to his volunteers.

An added benefit of this design is that it keeps complex algorithms out of the collector. Existing systems have devoted considerable effort to dealing with problems in provenance representation, such as compact storage or graph cycles [?]. Our design simply allows the handler to address these problems in whatever way is most appropriate.

4. SYSTEM-LEVEL OBJECT MODEL

Collecting system-level provenance requires a clear model of system-level objects. For each object, we must first describe how data flows into, out of, and through it. Next, we identify the LSM hooks (listed in Table **??**) which mediate data-manipulating operations on that object, or we place new hooks if the existing ones are insufficient. Finally, we decide how the relevant objects can be uniquely identified in the provenance log.

Each entry in the provenance log describes a single action on a kernel object. This includes the type of action, the subject, the object, and any appropriate context. For example, starting a kernel build could generate the following entry:

Type	Execute
Subject	Credential 508
Object	Root filesystem, inode 982
Arguments	"make", "–j8", "bzImage"
Environment	"HOME=/home/alice", "PATH=/usr/bin:/bin", "SHELL=/bin/bash", ...

For the purposes of recording provenance, each object which can appear in the log must be assigned an identifier which is unique for the lifetime of that object. Some objects, such as inodes, are already assigned a suitable identifier by the kernel. Others, such as sockets, require special treatment. For the rest, we generate a "provid," a small integer which is reserved for the object until it is destroyed. These provids are managed in the same way as process identifiers to ensure that two objects cannot simultaneously have the same provid. When an object which needs an identifier is created, we allocate a provid and attach it using the opaque `security` pointer provided by LSM. When the object is freed, we release the provid to be used again.

In later sections, we will show log entries in an abbreviated, human-readable form, with inode numbers resolved to filenames, and forks implied by a change in the bracketed provid:

```
[508] exec rootfs:/usr/bin/make -j8 bzImage
```

4.1 System, Processes, and Threads

Our model of data flow includes transferring data between multiple systems or multiple boots of a system. We therefore need to identify each boot separately. To ensure that these identifiers do not collide, we create a random UUID at boot time. We then write it to the provenance log so that subsequent events can be associated with the system on which they occur.

Within a Linux system, the only actors are processes[1] and the kernel. These actors store and manipulate data in their respective address spaces, and we treat them as black boxes for the purpose of provenance collection. Most data flows between processes use one of the objects described in subsequent sections. However, several actions are specific to processes: forking, program execution, and changing subjective credentials.

[1]On Linux, threads are a special case of processes, so we will use the term "process" to refer collectively to both.

Since LSM is designed to include kernel actions, it does not represent actors using a PID or `task_struct` structure. Instead, LSM hooks receive a `cred` structure, which holds the user and group credentials associated with a process or kernel action. Whenever a process is forked or new credentials are applied, a new credential structure is created, allowing us to use these structures to represent individual system actors. As there is no identifier associated with these `cred` structures, we generate a provid to identify them.

4.2 Files and Filesystems

Regular files are the simplest and most common means of storing data and sharing it between processes. Data enters a file when a process writes to it, and a copy of this data leaves the file when a process reads from it. Both reads and writes are mediated by a single LSM hook, which identifies the the actor, the open file descriptor, and whether the action is a read or a write. Logging file operations is then straightforward.

Choosing identifiers for files, on the other hand, requires some thought. We must consider that files differ from other system objects in that they are persistent, not only across reboots of a single system, but also across systems (like a file on a portable USB drive). Because of this, it must be possible to uniquely identify a file independent of any running system. In this case, we can make use of identifiers which already exist rather than generate new ones. Each file has an inode number which is unique within its filesystem. If we combine this with a UUID that identifies the filesystem itself, we obtain a suitable identifier that will not change for the lifetime of the file. UUIDs are generated for most filesystems at creation, and we generate random UUIDs for the Linux kernel's internal pseudo-filesystems when they are initialized. We can then use the combination of UUID and inode number to identify the file in all filesystem operations, as well as to identify a program file when it is being executed.

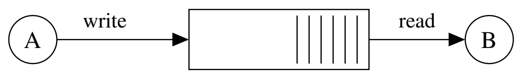
4.3 Memory Mapping

Files can also be mapped into one or more processes' address spaces, where they are used directly through memory accesses. This differs significantly from normal reading and writing in that the kernel does not mediate accesses once the mapping is established. We can only record the mapping when it occurs, along with the requested access mode (read, write, or both). Note that this does not affect our notion of complete mediation if we conservatively assume that flows via memory-mapped files take place whenever possible.

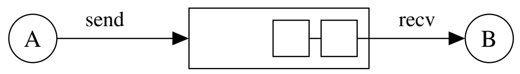
Shared memory segments are managed and interpreted in the same way. POSIX shared memory is implemented using memory mapping, so it behaves as described above. XSI shared memory, though managed using different system calls and mediated by a different LSM hook, also behaves the same way, so our model treats them identically. In fact, since shared memory segments are implemented as files in a temporary filesystem, their identifiers can be chosen in the same way as file identifiers.

4.4 Pipes and Message Queues

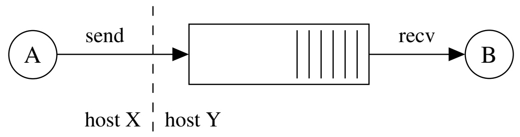
The remaining objects have stream or message semantics, and they are accessed sequentially. In these objects, data is stored in a queue by the writer and retrieved by the reader. The simplest such object is the pipe, or FIFO. Pipes have stream semantics and, like files, they are accessed using the `read` and `write` system calls. This interaction is illustrated in Figure **??**. Since a pipe can have multiple writers or readers, we cannot represent it as a flow directly from one process to another. Instead, we must split the flow into two parts, modeling the data queue as an independent file-like



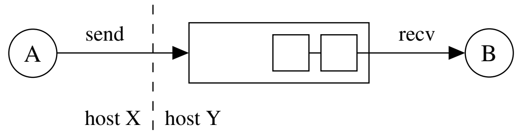
(a) Pipe



(b) Message queue



(c) Stream socket



(d) Message-oriented socket

Figure 3: Models of data flow

object. In this way, a pipe behaves like a sequentially-accessed regular file. In fact, since named pipes are inodes within a regular filesystem, and unnamed pipes are inodes in the kernel's "pipefs" pseudo-filesystem, we can choose pipe identifiers exactly as we do for files.

Message queues are similar to pipes, with two major semantic differences: the data is organized into discrete messages instead of a single stream, and these messages can be delivered in a different order than that in which they are sent. Fortunately, LSM handles messages individually, so we can create a unique identifier for each. We can then reliably tell which process receives it, regardless of the order in which the messages are dequeued. Since individual messages have no natural identifier, we generate a provid for each.

4.5 Sockets

Sockets are the most complex form of inter-process communication handled by our system, but they can be modeled very simply. As with pipes, we treat a socket's receive queue as an intermediary file between the sender and receiver, as shown in Figure **??**. Sending data, then, is just writing to this queue, and receiving data is reading from it. The details of network transfer are hidden by the socket abstraction, so we only need to consider the semantic differences between socket types.

Stream sockets provide the simplest semantics with respect to data flow: they behave identically to pipes. Since stream sockets are necessarily connection-mode, all of the data sent over a stream socket will arrive in the same receive queue. If we assign one identifier to each socket endpoint, we can use these identifiers for the lifetime of the socket. Message-oriented sockets, on the other hand, do not necessarily have the same guarantees. They may be connection-mode or connectionless, reliable or unreliable, ordered or unordered. We only know that any messages which are delivered are delivered intact. Each packet therefore needs a separate identifier, since we cannot be sure at what endpoint it will arrive.

In determining how identifiers are chosen, we must reason carefully about socket behavior. We should never reuse an identifier, since a datagram can have an arbitrarily long lifetime. We also want the identifier to be associated with the originating host. The per-boot UUID described in Section **??** addresses both of these requirements. By combining this UUID with an atomic counter, we can generate unique identifiers for socket provenance. As long as this counter is large enough to avoid rolling over, we can be reasonably certain that socket identifiers will remain unique.

In order to generate useful log entries, we must consider the sequence of events for sending and receiving data. Suppose process A on host X sends data which arrives in queue Q on host Y. Process B on host Y then receives this data. In this case, the following events should occur:

1. Process A passes data to the `send` function. X writes "A sends to Q" to the log.
2. The data is encapsulated in a packet as defined by the socket's protocol family.
3. The packet may be transmitted over a network.
4. The packet is either dropped, in which case no flow takes place, or it is delivered and saved to queue Q.
5. Process B is given some data from this queue as the output from the `recv` function. Before returning control to B, Y writes "B receives from Q" to the log.

Writing the "send" entry is the tricky step, because the sender needs to know the identifier of the remote receive queue. However, the sender and receiver may not have any shared information which can be used to agree on an identifier, so the cleanest solution is for the *sender* to choose an identifier for the remote receive queue and transmit it along with the first data packet. (How this happens depends on the socket's protocol family.) In this way, both the sender and receiver have the data needed to write their log entries.

5. IMPLEMENTATION DETAILS

In the course of creating Hi-Fi, we have overcome a variety of implementation challenges. Several of our solutions, such as running a provenance-opaque process, are new to the literature. Others, such as moving data efficiently from the kernel to userspace, are new solutions to problems that existing provenance work has solved in other ways.

5.1 Efficient Data Transfer

Provenance collection has been noted to generate a large volume of data [**?**]. Because of this, we need an efficient and reliable mechanism for making large quantities of kernel data available to userspace. Other systems have accomplished this by using an expanded `printk` buffer [**?**], writing directly to on-disk log files [**?**], or using FUSE [**?**]. However, none of these methods is appropriate for our system design. Instead, we use a Linux kernel object known as a "relay," which is designed specifically to address this problem [**?**].

A relay is a kernel ring buffer made up of a set of preallocated sub-buffers. Once the relay has been initialized, the collector writes provenance data to it using the `relay_write` function. This data will appear in userspace as a regular file, which can be read by the provenance handler. Since the relay is backed by a buffer, it retains provenance data even when the handler is not running, as is the case during boot, or if the handler crashes and must be restarted.

Since the number and size of the sub-buffers in the relay are specified when it is created, the relay has a fixed size. Although the collector can act accordingly if it is about to overwrite provenance which has not yet been processed by the handler, it is better to avoid this situation altogether. To this end, we allow the relay's size parameters to be specified at boot time.

5.2 Early Boot Provenance

The Linux kernel's boot-time initialization process consists of setting up a number of subsystems in sequence. One of these subsystems is the VFS subsystem, which is responsible for managing filesystem operations and the kernel's in-memory filesystem caches. These caches are allocated as a part of VFS initialization. They are then used to cache filesystem information from disk, as well as to implement memory-backed "pseudo-filesystems" such as those used for pipes, anonymous memory mappings, temporary files, and relays.

The security subsystem, which loads and registers an LSM, is another part of this start-up sequence. This subsystem is initialized as early as possible, so that boot events are also subject to LSM mediation. In fact, the LSM is initialized *before* the VFS, which has a peculiar consequence for the relay we use to implement the provenance log. Since filesystem caches have not yet been allocated, we cannot create a relay when the LSM is initialized. Our design goal of fidelity makes this a problem unique to our system: not only are we forced to postpone relay setup, but we must also do so without losing boot provenance data.

We therefore separate relay creation from the rest of the module's initialization and register it as a callback in the kernel's generic "initcall" system. This allows it to be delayed until after the core subsystems such as VFS have been initialized. In the meantime, provenance data is stored in a small temporary buffer. Inspection of this early boot provenance reveals that a one-kilobyte buffer is sufficiently large to hold the provenance generated by the kernel during this period. Once the relay is created, we flush the contents of the temporary boot-provenance buffer to it and free the buffer. By doing this, we can collect and retain provenance data for a large portion of the kernel's initialization process.

5.3 Operating System Integration

One important aspect of Hi-Fi's design is that the provenance handler must be kept running to consume provenance data as it is written to the log. Since the relay is backed by a buffer, it can retain a certain amount of data if the handler is inactive or happens to crash. It is important, though, that the handler is restarted in this case. Fortunately, this is a feature provided by the operating system's `init` process. By editing the configuration in `/etc/inittab`, we can specify that the handler should be started automatically at boot, as well as respawned if it should ever crash.

We also want to collect and retain provenance data for as much of the operating system's shutdown process as possible. At shutdown time, the `init` process takes control of the system and executes a series of actions from a shutdown script. This script asks processes to terminate, forcefully terminates those which do not exit gracefully, unmounts filesystems, and eventually powers the system off. Since the provenance handler is a regular userspace process, it is subject to this shutdown procedure as well. However, there is no particular order in which processes are terminated during the shutdown sequence, so it is possible that another process may outlive the handler and perform actions which generate provenance data. Our goal of fidelity requires that we collect this provenance.

Our solution is to handle the shutdown process in the same way we would handle a crash: restart the provenance handler. We modify the shutdown script to re-execute the handler after all other processes have been terminated, just before filesystems are unmounted. For this special case, we implement a "one-shot" mode in the handler which, instead of forking to the background, exits after handling the data currently in the log. This allows it to handle any remaining shutdown provenance, then return control to `init` to complete the shutdown process.

5.4 Bootstrapping Filesystem Provenance

Intuitively, a complete provenance record contains enough information to recreate the structure of an entire filesystem. To do this, we need to have three things: a list of inodes, filesystem metadata for each inode, and a list of hard links (filenames) for each inode. Our system has a hook corresponding to each of these items. Assuming, then, that we can collect provenance for a filesystem starting from the point when it is completely empty, all of this information will appear in the record.

There are two problems with this assumption, however. First, it is impractical. We may connect a USB drive which has been used elsewhere, or we may want to start collecting provenance on an existing, populated filesystem. Second, it is actually impossible to start with an empty filesystem. Without a root inode, which is created by the corresponding `mkfs` program, a filesystem cannot even be mounted. Unfortunately, `mkfs` does this by writing directly to a block device file, which does not generate the expected provenance data.

What we need is a way to bootstrap provenance on a populated filesystem. In order to have a complete record for each file, we must generate a creation event for any pre-existing inodes. We have implemented a utility called `pbang` (for "provenance Big Bang") which does this by traversing the filesystem tree. For each new inode it encounters, it outputs an allocation entry for the inode, a metadata entry containing its attributes, and a link entry containing its filename and directory. For previously encountered inodes, it only outputs a new link entry. All of these entries are written to a file to complete the provenance record. To make a new provenanced filesystem, we create it normally using `mkfs`, then run `pbang` immediately afterward.

5.5 Provenance-Opaque Flag

We noticed a strange behavior in the early prototypes of Hi-Fi: even when the system was completely idle, a continuous stream of provenance data was being generated. Inspection of the provenance record showed that this data described the actions of the provenance handler itself. The handler would call the `read` function to retrieve data from the provenance log, which then triggered the `file_permission` LSM hook. The collector would record this action in the log, where the handler would again read it, triggering `file_permission`, and so on. This created a large amount of "feedback" in the provenance record.

In light of our design goals, this is technically correct behavior. However, it floods the provenance record with data which does not provide any additional insight into the system's operation. One option for solving this problem is to make the handler completely exempt from provenance collection. This, however, has the potential to interfere with our ability to reconstruct the filesystem. If the handler were to create or move a file without generating provenance data, we could no longer accurately reconstruct the filesystem structure from the record. Instead, we make the handler "provenance-opaque," treating it as a black box which only generates provenance data if it makes any significant changes to the filesystem.

The first piece to our solution is informing the LSM which process is the provenance handler. To do this, we leverage the LSM framework's integration with extended filesystem attributes. We identify the provenance handler program by setting an attribute called `security.hifi`. The "security" attribute namespace, which is reserved for attributes used by security modules, is protected from tampering by malicious users. When the program is executed, the `bprm_check_security` hook examines this property for the value "opaque" and sets a flag in the process's credentials indicating that it should be treated accordingly. In order to allow the handler to create new processes without reintroducing the original problem—for instance, if the handler is a shell script—this flag is propagated to any new credentials that the process creates.

5.6 Socket Provenance

Our modifications to network socket behavior are designed to be both transparent and incrementally deployable. To allow interoperability with existing non-provenanced hosts, we place packet identifiers in the IP Options header field. In order to ensure that every packet sent by our system is marked appropriately, we implement two Netfilter hooks, which process packets at the network layer. The outgoing hook labels each packet with the correct identifier just before it encounters a routing decision, and the incoming hook reads this label just after the receiver decides the packet should be handled locally. Note that even packets sent to the loopback address will encounter both of these hooks.

In designing the log entries for socket provenance, we aim to make the reconstruction of information flows from multiple system logs as simple as possible. When the sender and receiver are on the same host, these entries should behave the same as reads and writes. When they are on different hosts, the only added requirement should be a partial ordering placing each send before all of its corresponding receives. Lamport clocks [**?**] would satisfy this requirement.

The problem with this is that the `socket_recvmsg` hook, which was designed for access control, executes before a process attempts to receive a message. This may occur before the corresponding `socket_sendmsg` hook is executed. To solve this, we place a `socket_post_recvmsg` hook after the message arrives and before it is returned to the receiver, and we use this hook to generate the entry for receiving a message.

We implement support for TCP and UDP sockets to demonstrate provenance for both connection-mode and connectionless sockets, as well as both stream and message-oriented sockets. Support for the other protocols and pseudo-protocols in the Linux IP stack, such as SCTP, ping, and raw sockets, can be implemented using similar techniques. For example, SCTP is a sequential packet protocol, which has connection-mode and message semantics.

5.6.1 TCP Sockets

TCP and other connection-mode sockets are complicated in that a connection involves three different sockets: the client socket, the listening server socket, and the server socket for an accepted connection. The first two are created in the same way as any other socket on the system: using the `socket` function, which calls the `socket_create` and `socket_post_create` LSM hooks. However, sockets for an accepted connection on the server side are created by a different sequence of events. When a listening socket receives a connection request, it creates a "mini-socket" instead of a full socket to handle the request. If the client completes the handshake, a new child socket is cloned from the listening socket, and the relevant information from the mini-socket (including our IP options) is copied into the child. In terms of LSM hooks, the `inet_conn_request` hook is called when a mini-socket is created, and the `inet_csk_clone` hook is called when it is converted into a full socket. On the client side, the `inet_conn_established` hook is called when the SYN+ACK packet is received from the server.

Our system must treat the TCP handshake with care, since there are two different sockets participating on the server side. We create a unique identifier for the mini-socket in the `inet_conn_request` hook, and this identifier is later copied directly into the child socket. The client must then be certain to remember the correct identifier, namely, the one associated with the child socket. The first packet that the client receives (the SYN+ACK) will carry the IP options from the listening parent socket. To keep this from overriding the child socket, we use the `inet_conn_established` hook to clear the saved identifier so that it is later replaced by the correct one.

5.6.2 UDP Sockets

Since UDP sockets are connectionless, we must use an LSM hook to assign a different identifier to each datagram. In addition, this hook must run in process context, so that we can record the identifier of the process which is sending or receiving. The only existing LSM socket hook with datagram granularity is the `sock_rcv_skb` hook, but it is run as part of an interrupt when a datagram arrives, not in process context. The remaining LSM hooks are placed with socket granularity; therefore, we must place two additional hooks to mediate datagram communication.

The construction and delivery semantics for UDP datagrams are not as straightforward as they may appear at first. An intuitive assumption would be that each datagram is constructed by a single process and received by a single process, but this is not the case. If the file descriptor of the receiving socket is shared between processes, they can all receive the same datagram by using the `MSG_PEEK` flag. In fact, multiple processes can also contribute data when *sending* a single datagram by using the `MSG_MORE` flag or the `UDP_CORK` socket option. Because of this, placing send and receive hooks for UDP is a very subtle task.

Since we consider each datagram an independent entity, the crucial points to mediate are the addition of data to the datagram and the reading of data from it. The Linux IP implementation includes a function which is called from process context to append data to an outgoing socket buffer. This function is called each time a process adds data to a corked datagram, as well as in the normal case where a single process constructs a datagram and immediately sends it. This makes it an ideal candidate for the placement of the send hook, which we call `socket_dgram_append`. Since this hook is placed in network-layer code, it can be applied to any message-oriented protocol and not just UDP.

We also place the receive hook in protocol-agnostic code, for similar flexibility. The core networking code provides a function which retrieves the next datagram from a socket's receive queue. UDP and other message-oriented protocols use this function when receiving, and it is called once for each process that receives a given datagram. This is an ideal location for the message-oriented receive hook, so we place the `socket_dgram_post_recv` hook in this function.

6. EVALUATION

The motivation behind this work is to determine whether whole-system provenance collection can provide useful information in a security context. We demonstrate this in two ways. First, we show that a number of typical malware behaviors appear plainly in a whole-system provenance record. In particular, when malware spreads from one provenanced host to another, we can observe the communication between the infected process on one host and the target process on the other using socket provenance. Second, we demonstrate that the performance overhead of Hi-Fi is small enough that it could be used in practice.

6.1 Recording Malicious Behavior

Our first task is to show that the data collected by Hi-Fi is of sufficient fidelity to be used in a security context. We focus our investigation on detecting the activity of network-borne malware. A typical worm consists of several parts. First, an exploit allows it to execute code on a remote host. This code can be a dropper, which serves to retrieve and execute the desired payload, or it can be the payload itself. A payload can then consist of any number of different actions to perform on an infected system, such as exfiltrating data or installing a backdoor. Finally, the malware spreads to other hosts and begins the cycle again.

For our experiment, we chose to implement a malware generator which would allow us to test different droppers and payloads quickly and safely. The generator is similar in design to the Metasploit Framework [**?**], in that you can choose an exploit, dropper, and payload to create a custom attack. However, our tool also includes a set of choices for generating malware which automatically spreads from one host to another; this allows us to demonstrate what socket provenance can record about the flow of information between systems. The malware behaviors that we implement and test are drawn from Symantec's technical descriptions of actual Linux malware[**?**].

To collect provenance data, we prepare three virtual machines on a common subnet, all of which are running Hi-Fi. The attacker generates the malware on machine A and infects machine B by exploiting an insecure network daemon. The malware then spreads automatically from machine B to machine C. For each of the malicious behaviors we wish to test, we generate a corresponding piece of malware on machine A and launch it. Once C has been infected, we retrieve the provenance logs from all three machines for examination.

Each malware behavior that we test appears in some form in the provenance record. In each case, after filtering the log to view only the vulnerable daemon and its descendants, the behavior is clear enough to be found by manual inspection. Below we describe each behavior and how it appears in the provenance record.

6.1.1 Persistence and Stealth

Frequently, the first action a piece of malware takes is to ensure that it will continue to run for as long as possible. In order to persist after the host is restarted, the malware must write itself to disk in such a way that it will be run when the system boots. The most straightforward way to do this on a Linux system is to infect one of the startup scripts run by the `init` process. Our simulated malware has the ability to modify `rc.local`, as the Kaiten trojan does. This shows up clearly in the provenance log:

```
[6fe] write B:/etc/rc.local
```

In this case, the process with provid 0x6fe has modified `rc.local` on B's root filesystem. Persistent malware can also add cron jobs or infect system binaries to ensure that it is executed again after a reboot. Examples of this behavior are found in the Sorso and Adore worms. In our experiment, these behaviors result in similar log entries:

```
[701] write B:/bin/ps
```

for an infected binary, and

```
[710] write B:/var/spool/cron/root.new
[710] link B:/var/spool/cron/root.new to
           B:/var/spool/cron/root
[710] unlink B:/var/spool/cron/root.new
```

for an added cron job.

Some malware is even more clever in its approach to persistence. The Svat virus, for instance, creates a new C header file and places it early in the default include path. By doing this, it affects the code of any program which is subsequently compiled on that machine. We include this behavior in our experiment as well, and it appears simply as:

```
[707] write B:/usr/local/include/stdio.h
```

6.1.2 Remote Control

Once the malware has established itself as a persistent part of the system, the next step is to execute a payload. This commonly includes installing a backdoor which allows the attacker to control the system remotely. The simplest way to do this is to create a new root-level user on the system, which the attacker can then use to log in. Because of the way UNIX-like operating systems store their account databases, this is done by creating a new user with a UID of 0, making it equivalent to the root user. This is what the Zab trojan does, and when we implement this behavior, it is clear to see that the account databases are being modified:

```
[706] link (new) to B:/etc/passwd+
[706] write B:/etc/passwd+
[706] link B:/etc/passwd+ to B:/etc/passwd
[706] unlink B:/etc/passwd+
[706] link (new) to B:/etc/shadow+
[706] write B:/etc/shadow+
[706] link B:/etc/shadow+ to B:/etc/shadow
[706] unlink B:/etc/shadow+
```

A similar backdoor technique is to open a port which listens for connections and provides the attacker with a remote shell. This approach is used by many pieces of malware, including the Plupii and Millen worms. Our experiment shows that the provenance record includes the shell's network communication as well as the attacker's activity:

```
[744] exec B:/bin/bash -i
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[751] exec B:/bin/cat /etc/shadow
[751] read B:/etc/shadow
[751] socksend B:173
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[744] link (new) to B:/testfile
[744] write B:/testfile
```

Here, the attacker uses the remote shell to view `/etc/shadow` and to write a new file in the root directory. Since the attacker's system is unlikely to be running a trusted instance of Hi-Fi, we see "unknown" socket entries, which indicate data received from an unprovenanced host. Remote shells can also be implemented as "reverse shells," which connect from the infected host back to the attacker. Our tests on a reverse shell, such as the one in the Jac.8759 virus, show results identical to a normal shell.

6.1.3 Exfiltration

Another common payload activity is data exfiltration, where the malware reads information from a file containing password hashes, credit card numbers, or other sensitive information and sends this information to the attacker. Our simulation for this behavior reads the `/etc/shadow` file and forwards it in one of two ways. In the first test, we upload the file to a web server using HTTP, and in the second, we write it directly to a remote port. Both methods result in the same log entries:

```
[85f] read B:/etc/shadow
[85f] socksend B:1ae
```

Emailing the information to the attacker, as is done by the Adore worm, would create a similar record.

6.1.4 Spread

Our experiment also models three different mechanisms used by malware to spread to newly infected hosts. The first and simplest is used when the entire payload can be sent using the initial exploit. In this case, there does not need to be a separate dropper, and the resulting provenance log is the following (indentation is used to distinguish the two hosts):

```
[807] read A:/home/evil/payload
[807] socksend A:153
    [684] sockrecv A:153
    [684] write B:/tmp/payload
```

The payload is then executed, and the malicious behavior it implements appears in subsequent log entries.

Another mechanism, used by the Plupii and Sorso worms, is to fetch the payload from a remote web server. We assume the web server is unprovenanced, so the log once again contains "unknown" entries:

```
[7ff] read A:/home/evil/dropper
[7ff] socksend A:15b
    [685] sockrecv A:15b
    [685] write B:/tmp/dropper
    [6ef] socksend B:149
    [6ef] sockrecv unknown
    [6ef] write B:/tmp/payload
```

If the web server were a provenanced host, this log would contain host and socket IDs in the `sockrecv` entry corresponding to a `socksend` on the server.

Finally, to illustrate the spread of malware across several hosts, we tested a "relay" dropper which uses a randomly-chosen port to transfer the payload from each infected host to the next. The combined log of our three hosts shows this process:

```
[83f] read A:/home/evil/dropper
[83f] socksend A:159
    [691] sockrecv A:159
    [691] write B:/tmp/dropper
    [6f5] exec B:/tmp/dropper
[844] read A:/home/evil/payload
[844] socksend A:15b
    [6fc] sockrecv A:15b
    [6fc] write B:/tmp/payload
    [74e] read B:/tmp/dropper
    [74e] socksend B:169
        [682] sockrecv B:169
        [682] write C:/tmp/dropper
        [6e6] exec C:/tmp/dropper
    [750] read B:/tmp/payload
    [750] socksend B:16b
        [6ed] sockrecv B:16b
        [6ed] write C:/tmp/payload
```

Here we can see the attacker transferring both the dropper and the payload to the first victim using two different sockets. This victim then sends the dropper and the payload to the next host in the same fashion.

6.1.5 Full Simulation

For a comprehensive test, we use our tool to implement a full simulation of the Linux Adore worm according to Symantec's description. Our provenance record captures the entire life cycle of the worm:

System call	Baseline	With Hi-Fi	Overhead
open	13.8	13.8	0.0%
close	10.6	10.7	1.0%
read	13.7	14.6	6.2%
write	21.4	21.3	-0.2%
creat	24.1	24.4	1.1%
rename	19.8	20.0	0.9%
unlink	36.4	36.7	0.7%
clone	74.6	74.0	-0.7%
execve	150.3	155.1	3.2%

Table 2: Mean execution time for system calls (µs)

- The compromised daemon downloading and extracting the payload tarball
- Execution of `start.sh`, which activates the payload
- Replacement of the `ps` binary with a trojaned version, and copying the original `ps` to `/usr/bin/adore`
- Installation of a cron job which kills the worm
- Replacement of `klogd` with a backdoor shell
- Emailing of the `/etc/shadow` file, process list, and network information to the attacker
- Infection of the next victim

We also successfully capture a sample backdoor session, in which the attacker views a user's command-line history and downloads an updated payload.

6.2 Performance

In addition to showing that Hi-Fi records malicious activity, we also wish to show that it does so without significantly degrading system performance. To this end, we benchmark a system running a stock Arch Linux kernel (version 3.2.13), then benchmark the same system with Hi-Fi compiled in. Our test system has two 2.30-GHz quad-core AMD Opteron processors, 16GB of RAM, and two 73GB hard disks in a RAID 0 array.

We first evaluate performance overhead at the system-call level using microbenchmarks. LMbench is frequently used for Linux microbenchmarks, but our initial results from this tool were inconsistent. Instead, we create a small program which exercises the major file and process operations. We then use the `strace` utility to measure the time spent in various system calls over a large number of executions of this program. The results of these benchmarks are summarized in Table **??**. For the system calls measured, the overhead is at most 6.2%, with most calls within 1% of the baseline.

To demonstrate the overall impact on system performance, we also run two macrobenchmarks customarily used in provenance system evaluation: a Linux kernel build, which evaluates a typical combination of process execution and file manipulation; and PostMark [**?**], which specifically stresses filesystem and disk transactions. We generate statistics from multiple executions of each benchmark using the Phoronix Test Suite utility [**?**]. With an unmodified kernel, our test system takes an average of 107 seconds to run the kernel-build benchmark. With Hi-Fi, this increases to 110 seconds, showing an overhead of only 2.8%. Performance on disk-heavy operations is unchanged, as PostMark achieves 2,083 transactions per second in both cases.

7. CONCLUSION

We have presented Hi-Fi, a system which applies the reference monitor concept to collect a high-fidelity provenance record suitable for security applications. We show that this record can be used to observe the behavior of malware, not only within a single host, but also across multiple provenanced hosts. Furthermore, we demonstrate that our implementation imposes less than 3% overhead on representative workloads and a similarly small overhead in system-call microbenchmarks.

We believe that Hi-Fi will provide a solid platform for future provenance research. For example, we do not explore options for working with provenance data after it is collected, but the modular design of Hi-Fi will make it simple to evaluate many different approaches to processing, storage, and querying. We have shown that complete system-level and socket provenance can provide deep insight into the design, performance, and security of systems and networks, and we believe that many other significant discoveries are yet to be made in this area.

Acknowledgements

This work is supported by the National Science Foundation under awards HECURA-0937944 and CNS-1118046.