Automatic High-Performance Reconstruction and Recovery

Ashvin Goel[+], Wu-chang Feng[*], Wu-chi Feng[*], David Maier[*]

Electrical and Computer Engineering, University of Toronto[+]
Computer Science, Portland State University[*]

Abstract

Self-protecting systems require the ability to instantaneously detect malicious activity at run-time and prevent execution. We argue that it is impossible to perfectly self-protect systems without false positives due to the limited amount of information one might have at run-time and that eventually *some* undesirable activity will occur that will need to be rolled back. As a consequence of this, it is important that self-protecting systems have the ability to completely and automatically roll back malicious activity which has occurred.

As the cost of human resources currently dominates the cost of CPU, network, and storage resources, we contend that computing systems should be built with automated analysis and recovery as a primary goal. Towards this end, we describe the design, implementation, and evaluation of Forensix: a robust, high-precision analysis and recovery system for supporting self-healing. The Forensix system records all activity of a target computer and allows for efficient, automated reconstruction of activity when needed. Such a system can be used to automatically detect patterns of malicious activity and *selectively undo* their operations.

Forensix uses three key mechanisms to improve the accuracy and reduce the human overhead of performing analysis and recovery. First, it performs comprehensive monitoring of the execution of a target system at the kernel event level, giving a high-resolution, application-independent view of all activity. Second, it streams the kernel event information, in real-time, to append-only storage on a separate, hardened, logging machine, making the system resilient to a wide variety of attacks. Third, it uses database technology to support high-level querying of the archived log, greatly reducing the human cost of performing analysis and recovery.

Key words: Self-healing computers, Computer forensics, Operating systems, Audit ing
PACS: 07.05.Bx

1 Introduction

It is envisioned that self-protecting and self-healing systems have the ability to perfectly identify intrusions and recover from them. We argue that 1) it is impossible to perfectly identify an intrusion while it is happening and 2) that in order to recover from one after it has happened, one requires a complete and usable log of *all* system activity. Consider, for example, an insider who logs in and starts deleting files in the file system. From a system's standpoint, it is impossible to recognize malicious intent until after a certain amount of activity is observed.

The goal of Forensix is to build a computer equivalent to "TiVo" that supports reliable reconstruction of all computer system activity for automatic and assisted forensic analysis and recovery. Its overall approach is to log all system activity of a target machine to a backend database. Queries are then issued on the backend to quickly determine the source of malicious behavior and to selectively "undo" such activity on the target machine. Using the case above, suppose the insider eventually performs an activity that finally indicates destructive intent, the Forensix system can then be used to automatically generate the entire activity tree of the session and use it to roll back *all* of the modifications.

To be effective, the system must gather an accurate, high-resolution image of system activities, sufficient for identifying a wide range of intrusions and answering questions such as "where did the attack come from", "what vulnerability was exploited", and "which files did the attacker modify". To support self-healing and self-recovery, the system must also be able to generate a selective "undo" log that allows the target system to be restored as if the intrusion never happened. In addition, the system activity log itself should be gathered in a tamper-resistant way, so that intruders cannot modify it or remove it to obscure their tracks. The collection mechanism should also not render the target system more vulnerable to non-intrusion based assaults such as denial of service attacks. Ideally, the system should have a small effect on the performance of the target system, and should be affordable in terms of its resource requirements. Finally, it should facilitate efficient and effective post-facto analysis, a process that is currently ad-hoc, time-intensive, manual and error-prone. In order to support such properties, this paper describes the design and implementation of *Forensix*: a high-resolution, analysis and reconstruction tool.

2 Motivation

Currently, when a system is compromised, investigators manually sift for clues based on the current state of the system and the log files that record the state of the system as it was under attack. This operation method is inherently "lossy", in that

vital information about where the hacker connected from, how the hacker entered and what the hacker did after he entered was not collected or may have been deleted by the hacker. This manual, error-prone process is unacceptable when considering the goals of automatic self-protection and self-healing. Consider a compromise in which the hacker has modified sensitive files to set up a backdoor into the system. Upon discovery, it would be ideal if system administrators could issue simple queries to the forensic system such as:

Query 1: Generate a list of sessions and processes that have written to the compromised files.

Query 2: Generate a system activity log for each session that was returned from *Query 1* in order to "undo" the activity.

There are many approaches for logging and auditing system usage, including application and system log files, process accounting mechanisms, network traffic traces, and file system checkers. While each has its strengths, none of them provide enough information by themselves to accurately recreate what happened in the system. For example, application and system log files only track events based on what the applications and system administrators think are necessary to log. Process accounting mechanisms only provide information as to how commands are executed and can fail to track what programs are doing internally. If a hacker downloads a binary onto the system and executes it, process accounting alone will not be able to show what the binary has done. For example, in the well-documented Mitnick case, a program called `zap2` was downloaded and compiled on the compromised system. The program was then executed multiple times in order to delete login entries from the system [26]. Network traffic traces alone are also problematic in that sessions are typically encrypted. In addition, even when they are not encrypted, they are targets for insertion and evasion attacks, thus making what has happened ambiguous [30]. It is also extremely difficult to correlate network forensic information directly to higher-level application behavior that elucidates the actual damage done to the target system. Finally, file system activity logs can only detect modifications to files and thus are unable to address attacks in which running processes are compromised directly [11].

3 Design Goals

To adequately perform analysis and recovery, the following goals must be met:

(1) **Completeness:** The system should collect and log enough information to completely capture user activity in order to efficiently reconstruct attacks. The system should also be able to glue the *who* (the user) and the *what* (all of the user's activities) together. Such a system needs to ensure that all activity is logged independent of system load. In addition, the system should be able

to support *fail-closed* operation when logging is compromised or disabled in order to prevent loss of any necessary logging information.

(2) **Authenticity:** No one should be able to spoof logging messages or tamper with the logging facility. Unlike the unauthenticated world of TCP/IP sessions, a strong authenticated relationship must be built between the logging facility and the storage system for the log data. The system should support logging immutability that prevents history from being rewritten. As seen in many cases, log files can be altered, which allows a hacker to change logging history and makes self-healing impossible.

(3) **Reproducibility:** The forensic system should allow users to accurately determine *who* and *what* for a wide variety of system activities such as incoming and outgoing network connections as well as files read or written by processes. It should allow correlating data based on time as well as system abstractions such as processes or sessions. The reconstruction process should be fast and should be independent of the length of time the system has been running.

(4) **Efficiency:** The amount of data collected and its encoding size should be minimized. Although one method for achieving the previous goal of completeness is a simple brute-force log of everything, this approach can hinder the ability to perform accurate, high-performance replay and reconstruction, even when the power and capacity of current hardware and software systems is fully leveraged. For self-healing to be practical, systems must be able to recover quickly to reduce the downtime of the system and increase availability.

4 The Forensix approach

Figure 1 shows the architecture of Forensix, a system that attempts to meet the design goals listed above. With Forensix, the *target system's* kernel is instrumented with a logging facility. In its current implementation, the logging facility streams system-call traces over a private network interface to a highly-secure *backend storage system*. While system call logging is prone to such problems such as race conditions, we are currently adapting our system architecture and approach to incorporate other, more accurate forms of logging such as logging within well-placed locations within the kernel and virtual-machine based logging [14]. This design is driven by the observation that a successful attack can only be caused by system-calls issued by processes running on the attacked system (provided the system is built correctly). Hence, if *all* system-call activity is captured and can be attributed to users, processes or connections, then it should be possible to accurately reconstruct *all* security incidents, immaterial of the type of attack. As a result, this approach helps satisfy our goal of completeness. In addition to completeness, system call logging provides compactness since Forensix does not record other, application-specific, events that do not impact system state. Other methods for improving compactness include data compression and suppressing system-call logging under certain con-

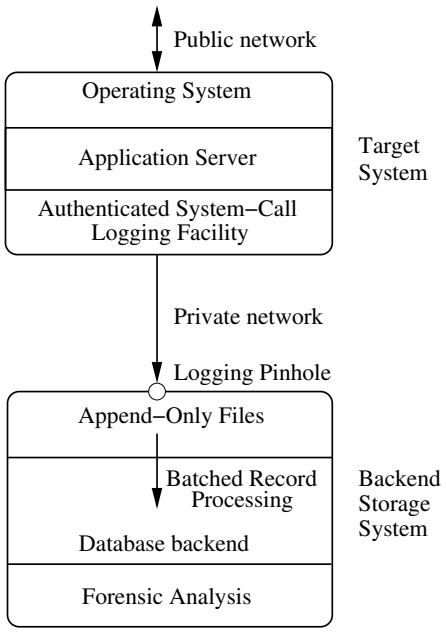

Fig. 1. Forensix system architecture.

ditions, such as reads to load common shared libraries.

For tamper-proof and immutable operation, Forensix logs system-call activity over a private network interface to a separate, append-only backend storage system with console-only login access. Immutability is achieved via the file system or via CD-R or DVD-R burning while tamper-proof operation is achieved by authenticating each target system at startup and by exporting only a minimum set of network services needed for securely logging system-call data. To support efficient and flexible querying, the backend periodically loads log data to a relational database. This forms the basis for accurate and high-performance replay. Queries are efficient because the database allows indexing frequently queried fields such as the user ID, the command executing the system call, and the starting time of the system call. In essence, the database holds a data warehouse for forensic analysis and query. While the amount of data being collected can be large, we argue that the system is feasible given the capacity of networking, CPU, and storage capacity available today. As a result, sacrificing some host and networking resources in order to add an automatic healing capability will be a fairly attractive proposition. The following subsections describe the logging facility and the backend storage system in more detail.

4.1 Kernel logging facility

To address the problems associated with the piecemeal logging approaches discussed in Section 2, Forensix logs within the kernel. In its current implementation, all activity across the system-call interface is captured and logged. By collecting all system-call activity and attributing this activity to individual connections and

sessions, the forensic backend will be able to recreate security incidents in an accurate, application and attack-independent manner. As attacks and attack signatures change, capturing activity at this point thus addresses the problem at a more fundamental, unified level. If the system is built correctly, the hacker will need to figure out a way to compromise a system without using a process, file, or connection in order to go undetected. For accurately attributing system activity to users, processes or connections, the key issue for the logging facility is the *type* and the *amount* of information needed.

The overall design of our logging system is founded on the notion that all intrusions start with a network connection or a console login, are processed by a daemon (`httpd`, `in.telnetd`, `in.ftpd`, `sshd`, `login`, etc.) and cascade into multiple system activities including other processes, file accesses, and outgoing connections. Our high-level goal is to assign these system activities to the initiating session, which helps to simplify and enhance the intrusion analysis and recovery process.

Figure 2 shows a diagram of various system activities and their relationships. The basic idea for capturing these relationships is to assign the identifier or the PID of the process that executes the activity as ownership information to each link of the graph. For example, incoming sessions, file accesses and outgoing connections are all associated with a process, while process creation via `exec` or `fork` is associated with the parent process.

For single-thread processes, this relationship attributes activities unambiguously. For example, one can derive the precise set of files accessed as a result of an incoming connection. Unfortunately, the relationship is more complicated for multi-threaded daemon processes. Consider a modern web server employing a process-mob architecture of pre-forked processes for handling requests. As several incoming sessions can be active at any one time, assigning ownership of a suspicious activity to a particular active session is difficult because threads can communicate via non-system call channels such as shared memory accesses. Forensix uses system-call tracing for achieving compactness, and while this approach may preclude complete disambiguation, the timing of an activity can be used as an effective discriminant. For example, the set of files accessed during the lifetime of a connection can be discovered. Similarly, the set of connections whose lifetimes were within the lifetime of a given connection and that accessed the previous set of files (the set of suspicious connections) can also be easily determined using the relationship graph. Based on the observations above, at a minimum, each system-call trace record has an associated PID and a time-stamp that helps to construct the activity relationship. Section 5 shows that this information allows constructing powerful forensic queries. In the future, we plan on examining other low-overhead mechanisms for unambiguously assigning ownership of activities to individual sessions.

Unlike previous approaches, which use system call sequences strictly for intrusion

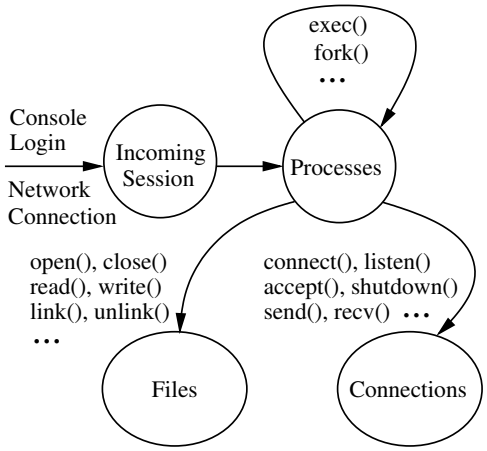

Fig. 2. The relationships between system activities.

```
execve("/bin/kill", ["kill", "11116"], [/* 62 vars */]) = 0
uname({sys="Linux", node="ren.cse.ogi.edu", ...}) = 0
brk(0)                                  = 0x804a9f4
open("/etc/ld.so.preload", O_RDONLY)    = -1
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=71060, ...}) = 0
old_mmap(NULL, 71060, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40014000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\260Y\1"..., 512)...
fstat64(3, {st_mode=S_IFREG|0755, st_size=1452984, ...}) = 0
old_mmap(0x42000000, 1290052, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0...
old_mmap(0x42134000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, ...
old_mmap(0x42139000, 8004, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MA...
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,...
munmap(0x40014000, 71060)               = 0
brk(0)                                  = 0x804a9f4
brk(0x804b9f4)                          = 0x804b9f4
brk(0)                                  = 0x804b9f4
brk(0x804c000)                          = 0x804c000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=30309872, ...}) = 0
mmap2(NULL, 2097152, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40027000
mmap2(NULL, 884736, PROT_READ, MAP_PRIVATE, 3, 0x19a) = 0x40227000
mmap2(NULL, 4096, PROT_READ, MAP_PRIVATE, 3, 0x298) = 0x40014000
close(3)                                = 0
kill(11116, SIGTERM)                    = 0
_exit(0)                                = ?
```

Fig. 3. `kill` system-call trace.

detection purposes [12,17,20,39,24,4,35], Forensix captures each system call, its timing, parameters, return values, the process issuing the call, and the owner of that process, throughout the lifetime of the server. This type of information is difficult to collect due to its size and semantic content. However, it is absolutely necessary, as described in the previous section, to recreate system activity.

To get an idea of the type and amount of information that can be collected, Figure 3 shows the system call trace that results when the `kill` command is performed on a process. While the trace is large, it is easy to identify the small number of system calls that clearly modify system state and should be logged (i.e. the initial

`execve` call and the second to last `kill` call). The trace also demonstrates the power of system-call traces over process accounting mechanisms. A wily hacker could download a binary implementing `kill` and name it something innocuous, thus avoiding detection. When logging system calls, it will become extremely difficult to hide such an activity.

It is clear that a limiting factor of our approach is the storage space for information being collected. As the capacity for processing and storing auditing information grows, the capacity of the server being traced and its network connection will as well. Fortunately, given the massive amount of unused local networking and storage resources and the ability to acquire such resources at relatively modest costs, we believe that the amount of data being collected is manageable. The fundamental reason for this is that it is clear that *Moore's law* governing advances in hardware systems is outpacing user and resource usage growth on the Internet. For example, consider a popular web site such as *http://slashdot.org*. While the site receives 50 million hits per month, its traffic growth has been outpaced by storage advances [38].

4.2 Backend storage system

The main job of the backend is to receive trace data from the network and store it in a form that allows issuing forensic queries. A simple form of storage is append-only log files. While such log files will contain all information needed for performing intrusion analysis, they will not necessarily be in a form amenable for efficient searching and manipulation. For example, we anticipate the need to make forensic queries such as

- Show all user sessions that executed `/bin/sh` from daemon processes other than `sshd`, `telnetd`, or `login` and group sessions by user.
- Show all activity for a particular user session *S*, specified by a source IP address and port, a user ID, and a connection timestamp.

To get some degree of efficiency, it is desirable to index data and ideally provide complete DBMS query processing capabilities to run the types of queries described above. To do so, Forensix stores the trace data in a relational database. While the keys chosen for building indexes depend on the types of queries that are likely to be executed, we have identified three candidate keys, *process ID*, *time* and *incoming connection identifier*, based on our model of attributing ownership to activities.

In addition to fast queries, the backend must provide high throughput storage or else it can become a bottleneck for the target system. The reason is that Forensix ensures that logging information is not lost, i.e. *fail-closed* behavior, by matching target system performance to the ability to log data and blocking the target system when the backend is unable to keep up. A naive approach for building the backend

is to continually insert records from the log files into the database. However, this approach places restrictive limits on rates that log file data can be absorbed, particularly because of the indexing overhead typically seen on multiple, small updates. To address this problem, we use bulk loading facilities available in most DBMSs for inserting large batches of traces with deferred indexing. Our experiments show that this approach removes much of the cost of fine-grain index updates. We are currently examining ways to avoid copying data from the log file to the database during loading by constructing log files so that they can be directly mapped into the data space of a specific DBMS.

5 Implementation

Forensix has been implemented on Linux and is freely available [1]. The implementation consists of 1) an auditing module and a sender daemon running on the target system, 2) a receiver daemon and a database injector running on the backend, and 3) database queries and scripts that allow replay of system activities for forensic purposes. Each is described below.

5.1 Target system

The auditing module of the target system consists of a Linux kernel module that traps system calls and logs data in a kernel buffer. The module code hijacks the system call table and adds stub code around several system calls to capture the system call, its timing, its parameters, its return value and the PID of the process making the call.

Based on our model of attributing system activity (see Figure 2), the system calls traced fall broadly in the three categories: networking, process management and file system. Network calls include such calls as `connect`, `accept`, `listen` and `shutdown`. Process management calls include `fork`, `exec`, `kill`, `exit` and `setuid`. Important file system calls include `open`, `read`, `write`, `close`, `symlink`, `link`, `mount`, `unmount`, `dup` and `chown`.

For tracing, Forensix adds stub code around system calls but does not change the calls themselves. This approach allows building the auditing code as a separate module but can introduce race conditions so that system activity cannot be completely reconstructed [18]. For example, a race condition exists between writing to a file *A* via a symbolic link and modifying the symbolic link to point to a different file *B*. Our stub approach may not accurately capture whether the file *A* or the file *B* was written because the precise timing of the two operations (writing the file and modifying the symbolic link) is not known to the tracing system. A solution to this

problem is to capture the output of pathname resolution while reading the symbolic link during the write operation. While this solution is simple and similar techniques can in principle be applied to resolve most timing-related race conditions, the code will be more intrusive than our auditing code.

The sender daemon is a kernel process that reads data from the kernel buffer and sends it over a private network to the backend. For fail-closed behavior, if this process is unable to retrieve or send data, then the auditing module stops system activity when the kernel buffer becomes full. This approach should provide automatic hardening provided kernel code can be trusted more than user-level code. Finally, the kernel buffer is statically sized based on the total available memory in the system, such as 10% of memory. While we believe that this technique will work well for most systems, other sophisticated approaches, such as buffer tuning for TCP sockets [37], could be applied in the future if the buffer often becomes a bottleneck.

5.2 Backend

The receiver daemon on the backend is a simple process that reads data from the network and stores it to human-readable, tab-separated, log files that is periodically loaded into a database. In Forensix, the database is optimized 1) for bulk loading (with index generation) and 2) for queries. In particular, data is read-only after it has been loaded and thus transactional guarantees are not essential.

The database stores several tables for the system call traces. The main table is called `events`, which stores common attributes, such as id, time, PID and return value, of every system call event. Data from system calls that is unique to specific calls is stored in separate tables to reduce redundancy and minimize the chances of inconsistency. Examples of such tables include `io`, `dup` and `connections`. The `io` table stores all reads and writes, while the `dup` table stores file opens, closes and file descriptor duplications. The `connections` table stores network-related system calls. In addition to these basic tables, Forensix constructs special database tables upon batch insertion to accelerate subsequent query processing. These "interval" tables precompute time-based attributes of files, connections, and processes that are common to many of the queries being performed.

5.3 Queries

In order to be useful, a powerful set of queries must be supported for post-facto analysis. Table 1 lists some examples of queries we have implemented. Simple queries are implemented using SQL directly.

Table 1
Examples of Forensix queries.

Query Name	Arguments	Output
Active_Processes	start_time, end_time	List all active processes within a given time interval.
Immediate_Children	PID	List all immediate children of a process.
Children	PID	List all children of a process.
Immediate_Parent	PID	List immediate parent of a process.
Parents	PID	List all parents of a process.
FDs_written	PID, start_time, end_time	List all file descriptors written by a process within a given time interval and the time they were written.
All_FDs	PID, filename, fd_list, time	List all file descriptors that refer to a filename or to other file descriptors in fd_list at a given time.
Did_Process_Write	PID, filename, start_time, end_time	Did process write to filename within a given time interval?
Writers	filename, start_time, end_time	List all processes that wrote to filename within a given time interval.
IO	PID, fd_list	List the timing and the data for I/O performed on file descriptors in fd_list by a process.
Replay_Shell	PID	Run IO query on file descriptors 0, 1 and 2 for a shell process.

6 Evaluation

A viable auditing and replay system should have low auditing overhead, reasonable space requirements and should be able to replay system activity in near-time. Hence, to evaluate Forensix, we performed two types of experiments that measure the performance and space overhead induced by auditing and the time taken to run queries. To measure system overhead, we ran two benchmarks on the target system: 1) Linux kernel build and 2) Webstone. The kernel build benchmark is mainly CPU

Table 2
Kernel build times.

	Auditing off	Auditing on Network off	Auditing on Network on
Total Time	233.2 s	247.1 s (6%)	252.0 s (8%)
System Time	14.0 s	26.3 s	30.7 s

The total time represents the time to complete compilation of the Linux kernel. The numbers in parenthesis represent the increase in completion time under Forensix versus standard Linux.

bound and does not stress the system much. However, it determines the viability of Forensix when running similar applications in a regular desktop environment. The second benchmark, Webstone, stresses a web server and is representative of a loaded server environment.

Our experiments were run on 1.8 GHz Intel Pentium-4 processors with 1 GB of memory. Both the target and the backend machines had the same configuration. In addition, for Webstone, the client process was run on a third similar machine. All the machines are connected with a gigabit network. The connection between the target machine and the backend machine was on a separate VLAN so it was not affected by other traffic, such as the client to target machine traffic during the Webstone benchmark. All machines run Redhat Linux 2.4.20 with the `ext3` file system and the target machine runs the Forensix auditing module. The backend machine uses the MySQL version 3.23 database.

6.1 Target system

Table 2 shows the results of the kernel build benchmark. The base result for building a kernel under Linux without Forensix auditing is shown under the "Auditing off" column. The second "Auditing on, Network off" column shows the results when auditing is turned on in the kernel and the sending daemon retrieves data from the kernel but does not stream it to the backend. In the final column, data is also streamed to the backend and stored in log files. The numbers in the table are generated by running the `time` command on the kernel build process.

The table shows that the benchmark completion time in our unoptimized implementation increases by 6% when auditing and by 8% when auditing and transmitting data. We believe that this overhead is a small price to pay for the ability to accurately and systematically reconstruct system state to capture the increasing number of system compromises we see today. Note that, as expected, almost all the additional time is spent in system activity.

Table 3
Webstone throughput.

	Auditing off	Auditing on Network off	Auditing on Network on
Throughput (Mb/s)	296.8	276.2 (93%)	186.87 (63%)

The Webstone benchmark stresses a standard Apache web server running on the target system by issuing back-to-back client requests. Figure 3 presents the key results for this benchmark, the throughput achieved by the web server. All the Webstone tests were run for approximately 36 minutes. The "Auditing off" column is the base throughput under Linux without Forensix auditing. The next column shows the throughput when auditing data and retrieving it from the kernel. The decrease in throughput in this case is 7%, which is similar to the overhead observed earlier for the kernel build benchmark.

The final column shows the result when data is also streamed to the backend and stored in log files. In this case, the throughput decreases by as much as 36%. Currently, we are in the process of profiling the kernel to investigate the reasons for this decreased throughput. However, we believe that there are two obvious optimizations that will help improve our results. First, our implementation is unoptimized and uses a very simple memory allocation mechanism for storing trace data. We expect that improving the auditing module's memory allocator will significantly reduce performance overhead. Second, for simplicity, the auditing module copies code from the kernel to the user space which is then copied back to the kernel to be sent to the backend. To minimize copies, data can be sent to the backend directly from the kernel. This optimization will also help reduce pressure from the memory subsystem.

6.2 Backend system

To evaluate the throughput of the database, we measured the row insertion rate of the database, i.e. the actual number of rows that can be inserted per second in the database. For the Webstone log files, the MySQL database could be bulk loaded at approximately 7400 rows/second. We also measured the row generation rate or the number of rows that are generated per second as data is captured in log files in real-time. For the Webstone test, the row generation rate is 17900 rows/second. This result indicates that for near-time intrusion analysis, where database loading takes less time on average than data generation, the web server can be heavily loaded for no more than 40% time during the day. We expect that this limitation will not be a problem in practice because of typical diurnal server activity [8].

Next, we measured the space requirements of the compressed log files for the kernel

build and the Webstone benchmarks. For the kernel build benchmark, the log files grow at 8.8GB/day, while for the Webstone benchmark they grow at 30GB/day. There are several reasons that these numbers are significantly larger than comparable data generated by ReVirt [14]. The first is that, unlike Forensix, ReVirt does not log filesystem I/O, relying instead on periodic checkpoints whose storage costs are not reported. Moreover, if checkpoints are infrequent, then replaying system activity for forensic analysis can take a long time, as much as the time period since the last checkpoint. The second reason is that we use a Gigabit network in our Webstone experiments and thus produce much more data than the 100 Mb/s network used in evaluating ReVirt. Normalizing for network speed, the Webstone log-file growth rate for Forensix is comparable to ReVirt.

6.3 Queries

In order to be useful, queries must be efficiently supported in near real-time. For evaluation, the Webstone benchmark was re-run and at the same time a user edited the `/etc/passwd` file on the target machine. We executed the `Replay_Shell` query (which uses the `IO` query, see Table 1) with the PID of the shell process in which the password file was modified. This complex query took under 10 seconds to run under MySQL, which we believe is a reasonable time to replay this system activity.

7 Forensix in practice

In this section, we describe results from experiences in using Forensix on a honeypot target system that was attacked multiple times during the course of a week. Then we show that our analysis tools that can be used interactively even on large data sets. Finally, we evaluate the performance overhead of the system and show that complete auditing imposes a small performance overhead and it is economically feasible to store all audit data for several months.

7.1 Setup

Our honeypot setup consists of a target and a backend machine both running AMD Athlon MP 2600+ machines with 512 MB RAM. The target runs stock RedHat 7.2 that has well-known vulnerable services including Apache httpd with SSL, Wu-ftpd, Sendmail, SAMBA and the ptrace exploit. We used the Snort network intrusion detection tool to detect potential intrusions. The backend machine is connected to the target on a separate network and has a firewall with a single open port that

```
/bin  74   /bin/kill 05-12 17:11:58
           /bin/ps 05-12 17:11:46
/dev  3
/etc  84   /etc/passwd 05-12 17:11:20
/home 11
/lib  588
/root 3    /root/.bash_history 05-12 18:40:32
/sbin 175  /sbin/ldconfig 05-12 17:12:09
/tmp  26
/usr  26   /usr/bin/killall 05-12 17:11:46
/var  452
```

Fig. 4. File-accesses for ftpd attack.

only allows a single connection from the target machine. The backend machine uses the MySQL version 4.1.10 database.

The target was run with the vulnerable services for approximately a week from May 11th until May 18rd, 2005. During this time, the target was attacked externally with the Wu-ftpd remote root exploit around 5pm on May 12th. We shut down the machine later that evening and reinstalled a new target system next morning.

7.2 Analysis of Ftpd Attack

In a typical ftpd intrusion, a remote attacker gains root access to the vulnerable system. On May 12 around 17:10 Snort reported an anonymous FTP login followed by command overflow attempts that contained shellcode. While Snort helps with detecting attacks, it provides little information about what actually happened on the system. To look for any recent changes in the file system, we ran a file-access tracking query to list all the files or directories modified between 17:00 and 19:00 of that day. A partial report, shown in Figure 4, lists the modified files grouped by root directories and their last modification times. The numbers in the second column show the number of modified files. Based on this report, we suspected that a rootkit had been installed.

Next we ran queries to calculate a dependency graph using the modified /usr/bin/killall (shown above) as one of the detection points. A partial resulting graph is shown in Figure 5. It shows the bash process that was spawned by the ftp daemon, the use of the passwd command and downloading of the rk.jpg file.

We then queried the backend for any instances of /dev/pts/x creations between 17:00 and 19:00. This query returned one row that showed that an interactive shell was used from 17:12 until 18:40. A query to obtain the process owner showed that this attacker's shell was run as root. Next, we used queries to obtain IO data in order

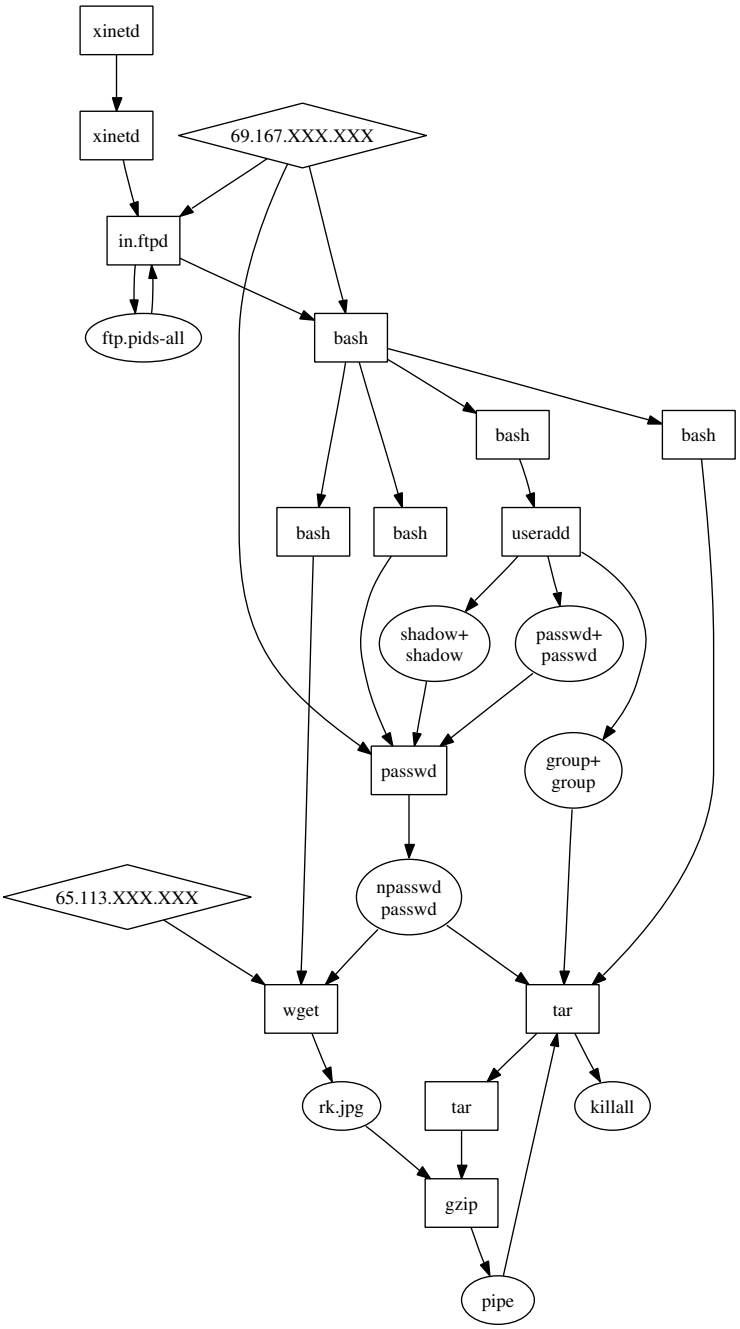

Fig. 5. Tracking the FTP intrusion.

to replay the shell. Key output from the shell session is shown in Figure 6.

Using our IO tracker, we recreated the removed psyBNC.tgz file that acts as both an IRC bot and an IRC bouncer or anonymizer. Its executable files are disguised as crond. The attacker runs the SucKIT rootkit that is loaded through /dev/kmem and does not need a kernel with support for loadable kernel modules [34]. With the rootkit, the attacker tried to hide the fake crond process but since we use LIDS [46] on the target system to disable writes to /dev/kmem, the attacker was not successful.

```
[root@rex www]# ftp -v 65.113.XXX.XXX
Name: XXXXXXX
Password:
get psyBNC.tgz
[root@rex www]# tar xzvf psyBNC.tgz
[root@rex www]# rm -rf psyBNC.tgz
[root@rex m4a1]# crond
Listening on: 0.0.0.0 port 6001
Thu May 12 17:18:11 :psyBNC2.3.1-cBtITLdDMSNp started (PID
:3975)
[root@rex .sk12]# ./sk i 3975 [= SucKIT version 1.3a, Jan 27
2005 =]
Can't open /dev/kmem for read/write (1)
[root@rex www]# w
6:40pm up 4:20, 0 users, ...
[root@rex log]# pico /var/log/messages
[root@rex www]# logout
```

Fig. 6. Shell replay for the ftpd attack.

Based on the dependency tracker graph, we recreated the removed rk.jpg file that installs a backdoor and then clears its traces from log files. To find out more about the backdoor, we issued a query on the connection and the process interval tables to find out about open ports between 17:00 and 18:00 and found a process called sendmail that was listening on port 212 from 17:12 and was used to run the interactive shell. Based on the analysis of this attack, it seems to be similar to the report available from the Honeynet Project [45].

7.3 Analysis Results

The total time taken to run each of the queries described above is shown in Table 4. This table shows that all these queries run quickly and can be used by an interactive user.

8 Self-healing via selective recovery

In this section, we describe the design of our selective-recovery tools that have been built over the Forensix audit system described in the previous sections. Selective recovery is necessary for self-healing systems so that they can keep running with minimal disruption after a compromise or other damage. The selective recovery tools we have built recover file system data after a remote intrusion or after local damage has occurred (e.g., management errors, disgruntled employees). This approach al-

Table 4
Time taken for each query.

Ftpd attack analysis	Time taken
List all the modified files and directories	20 s
Find root-owned setuid files that were executed by non-root processes	7 s
Dependency graph generation	25 s
Finding the interactive shells	< 1 s
Finding uid of the shell process	< 1 s
Replaying attacker's shell	1 s
Recreation of the removed attack files	3 s
Finding execves issued by children of compromised in.ftpd	1 s
Finding the listening port set by the attack code	< 1 s

lows systems to be more resilient to intrusions [2]. Selective recovery is the process of undoing "tainted" file-system modification activities while preserving legitimate activities. It has similarities with system software upgrade where the upgrade can be selectively rolled back without affecting the rest of the system [6,16].

Selective recovery presents two main challenges. First, the set of tainted file-system activities, or activities that were "affected" by the malicious activity, must be determined. Unlike with software upgrades, where the set of files associated with the upgrade is known, tainted activities can occur at arbitrary locations and can be the result of direct damage from an authorized user account or from an intrusion. Furthermore, determining such tainted activities can be difficult as they may be performed directly by the malicious entity or indirectly via a legitimate process that is vulnerable to attack. For example, consider a web server that services a malicious connection that in turn sets up a back-door to the system. The attacker then logs into the system via the back-door. If the web-server's activities are marked as tainted forever [23], then every future server activity will be tainted even though the server performs logically distinct and unrelated activities. Ideally, only the file-system activities related to the malicious connection and the back-door should be undone.

The second issue with selective recovery is that certain legitimate activities can depend on tainted activities. For example, a tainted activity may create a file that is later modified by a legitimate activity. We term such activities as "conflicting". With selective recovery, the challenge is to preserve legitimate file-system modification activities, minimize or isolate the ill-effects of conflicting activities, and at the same time, automate recovery as much as possible.

8.1 Selective recovery approach

The Forensix logs relate file-system activities to processes and allow analysis tools to replay execution. Using this rich source of audit information, a dependency analyzer is then run to create dependencies between processes, files, and sockets. These dependencies are based on the activities performed. This analysis and an initial set of externally provided tainted processes, files or sockets (e.g., either by an administrator or an intrusion detection system) helps derive the set of tainted activities. Finally, this set is used to selectively undo the effects of tainted file activities.

The choice of the set of tainted activities involves an inherent tradeoff in selective recovery. This set can be chosen conservatively, which simplifies the recovery process, but can mistakenly mark legitimate activities as tainted, causing legitimate data to be lost. In contrast, an optimistic choice helps preserve legitimate activity, but can miss tainted activities, thus making recovery less effective. Our analysis exposes this tradeoff by providing a choice of dependency policies, from conservative to optimistic. The most conservative policy taints all data after an attack and hence recovery leads to a snapshot of the file system before an attack. The optimistic policies recover data selectively by either limiting or ignoring dependencies. For example, we can optimistically assume that the web server's activities across different connections are unrelated and explicitly limit dependency within the server process to a certain time interval based on the connection it is serving.

The optimistic analysis policies lead to conflicts during recovery where legitimate activities that need to be preserved may "depend" on tainted activities that are undone. To enable automatic conflict resolution during recovery, we separate file-system activities into name, content and attribute activities and apply recovery actions to each type of activity separately. This approach simplifies resolution, allows recovery actions that are suited for each type of activity, and enables dealing with name and attribute conflicts completely automatically.

8.2 Example scenarios

We evaluated the functionality of the selective recovery system using several different attack or local damage scenarios that are briefly described below. For each scenario, we also describe the correct recovery action that should be taken. We then carried out these scenarios with Forensix logging turned on and used the logs to compute the set of recovery actions. We report in Section 8.3 on how the calculated recovery actions correspond to the stipulated recovery actions.

8.2.1 Illegal storage

A guest user logs into the system and executes the sendmail attack to get a root shell and creates a new root account by directly writing to the /etc/passwd and the /etc/shadow files. Later, the attacker logs in via this new account and changes the /etc/ftpaccess file so that anonymous users can create or delete files in the system. Finally, he uploads 500 illegal pictures into the system as an anonymous user.

Correct recovery action: Remove all the pictures, the files in /var/spool/mqueue that are generated by the sendmail attack and the home directory of the attacker's root account. In addition, the legitimate versions of the /etc/ftpaccess, /etc/passwd and /etc/shadow files need to be recovered.

8.2.2 Unhappy student

An attacker launches a remote attack on the wu-ftpd daemon running on the system. The attacker gets a root shell and downloads, compiles and installs a bindshell back-door. In addition, the attacker modifies a professor's home directory to be globally writable. Later, student *A* with a regular account creates a file in his own directory and then replaces the professor's grades file with a new file. Then student *B* copies the modified grades file to his own home directory.

Correct recovery action: Shut down the back-door process, recover the original grades file in the professor's directory, restore the attributes of the directory and the file and remove the copy of the file in student *B*'s home directory.

8.2.3 Content destruction

A software developer has been working on the files src/project.c, hfiles/p1.h and hfiles/p2.h. He has also saved a backup of the C file in backup/project.c.bak. Another developer on the system launches the pwck local escalation exploit to get a root shell. This attacker deletes the project.c and p2.h files. The victim notices that the project.c file is missing. He copies the backup file and moves the p1.h file to the src directory. Then, he deletes the hfiles directory and notifies the administrator.

Correct recovery action: Restore the hfiles directory, restore p2.h file into this directory, recover the original project.c file and deal with the two different versions of this file (the one copied by the user and the original deleted version).

8.2.4 Software installation

The next two scenarios present and analyze system administration errors. Using a root account, we installed the Realplayer 8 in the wrong directory which caused

it to create many files and subdirectories in this directory. In addition, it created or updated various Netscape, KDE and Gnome configuration files or directories in /root.

Correct recovery action: All the Realplayer files and subdirectories should be removed and the configuration files should be restored.

8.2.5 Inexperienced administrator

We download Gallery, a popular web-based photo album application. The Gallery administrator creates two accounts, one for himself and the other for a guest with a easily guessable password. The administrator then adds new albums and pictures to the website. Concurrently, an attacker logs in by correctly guessing the password of the guest account from a different remote site. The attacker creates a new album and a few pictures in this new album and visits the administrator's album. Then the administrator visits the attacker's album and detects a problem.

Correct recovery action: Remove the attacker's album and all related data (e.g., thumbnails) generated by Gallery.

8.3 Selective recovery evaluation

In all cases above, our selective recovery system produces less than two false positives (legitimate activity is marked tainted) or negatives (tainted activity is not caught) even though the number of recovery actions ranges between 5 to 1200. Given this positive result and the widely different scenarios and the recovery actions described above, we believe that our selective recovery system forms a good basis for a self-healing tool.

9 Related Work

This work consists of two main components, analysis and recovery. We focus on related work in these areas in turn.

9.1 System analysis and replay

System call traces have been used in the past to identify normal system behavior and then to automatically detect suspicious behavior or intrusions [20,36,39]. However, these approaches examine system-call patterns over a short window of 5-100

calls and are insufficient for completely capturing system activity for forensic purposes. In contrast, Forensix captures system calls, their timing, their parameters, their return values, the process making the call and their owners throughout the lifetime of the target system for accurate replay.

Forensix enables the off-line execution of techniques similar to those found in the STAT and USTAT systems which employ state transition diagrams to identify suspicious activities [21,15,28]. Forensix differs from these systems in that the auditing is done within the kernel at the system call level and the audit trail is securely transfered to an append-only backend storage system. The information being gathered is thus a super-set of that collected by the audit records in USTAT and is stored remotely in a secure manner. It should be possible to take the system call records and recreate the audit records of USTAT at the database backend and to run USTAT along with other intrusion analysis tools such as Tripwire [22]. In addition, because the information itself is archived, the information can be re-analyzed as additional knowledge is gained on specific intrusions.

Our analysis tools are directly motivated by the BackTracker [23] that uses a time-based approach to generate dependencies between processes, files and sockets and uses the dependency graph to view intrusions. The primary difference between the two stems from the difference in their goals. While the BackTracker is focused on tracking the sources of an intrusion, our analyzer generates a set of tainted files that need to be recovered. As a result, the BackTracker's tainting policies are conservative or else it would miss the intrusion, while we provide optimistic policies so that data can be preserved during recovery as much as possible. In addition, our optimistic policies use interval-based analysis and dependency sources to limit the effects of tainting. Another difference is that BackTracker does not provide precise details about all of the system activities. For example, it would show the steps that led to the modification of a sensitive password file, but does not show the precise changes made to the file. For the latter information, BackTracker must be used in combination with ReVirt, which places the system within a virtual machine and logs the VM-to-host instruction system. The clear advantage of ReVirt is that it removes non-determinism by serializing all system activity at the logging point and hence allows complete system replay. Another advantage is that the virtual machine approach does not require kernel integrity. However, unlike Forensix, ReVirt cannot support arbitrary queries without forcing the user to replay the entire instruction stream. On a heavily loaded system, such replay requires time that is proportional to the length of time the system has been running since the last checkpoint. Since forensic analysis is often an iterative process, such an approach defeats the initial goal of our work in reducing the time and human overhead required to perform forensic analysis.

Garfinkel [18] discusses the problems associated with system call interposition based security tools. Many of the problems described, such as argument races, occur due to user-level interception and do not exist in Forensix where auditing occurs

within the kernel. However, an important problem is understanding the complex Unix API and its side effects so that queries can be implemented correctly. Another problem is race conditions due to time-of-check/time-of-use bugs [5]. The main one we identified was traversal of symbolic links and relative pathnames during file system operations. Both can be solved by capturing the output of pathname resolution while reading the symbolic link during the file system operation. Quinlan and Dorward discuss a novel approach for storing append-only archival data [32] in Venti. Like in various peer-to-peer storage systems [13,10], data blocks for archival storage in Venti are identified by a collision-resistant hash, which eases the secure implementation of append-only storage. Such an approach could be used for the Forensix backend.

9.2 System recovery

Magpie [3] extracts the control flow and the resource requirements of requests in a clustered server environment by monitoring kernel and application-level events and correlating these events using an application-specific event schema. Magpie uses interval-based correlation similar to our dependency analysis. However, while Magpie uses undirected dependencies to clustered sets of events, our analysis uses directed dependencies to derive data flow. Data lifetime analysis using system-level simulation [9] or hardware-based information flow [42] allows detecting or protecting programs against malicious software attacks by identifying spurious information flows from untrusted I/O sources. Both can provide much more accurate dependency analysis than our approach but either run orders of magnitude times slower or require special architectural support.

Versioning file systems retain earlier versions of modified files, allowing recovery from user mistakes or system corruption. A key focus of versioning systems is encoding efficiency. For example, the Elephant file system [33] uses a clever purging method that keeps "landmark" data versions and purges generated and temporary files aggressively, while CVFS encodes metadata versions efficiently [41,40]. Our system, which uses a unoptimized data storage mechanism, would benefit from some of these techniques, although purging data versions would limit some of the benefits of selective recovery. While versioning approaches provide the basic capability to rollback system state to a previous time, such a rollback discards all modifications made since that time, regardless of whether they were done by a tainted or legitimate process.

The Repairable File System [47] has goals closest to our work. Its contamination analysis is similar to our dependency analysis although it only uses a propagation phase and does not have a notion of dependency intervals. In addition, the recovery phase does not seem to consider conflicting activities. Application-specific conflict resolution has been extensively studied in the context of replicated file sys-

tems [31,25] and databases [44]. While we have not experimented with these policies, they would directly apply to our recovery techniques.

Fastrek [27] applies selective recovery to databases by attributing modifications to malicious activities and then rolling back changes selectively. A potential issue with this approach is cascading aborts where a legitimate activity is rolled back if it may have depended on the data produced by a tainted activity. While conservative dependency policies in our system effectively achieve the same result, our conflict resolution policies allow using optimistic policies that reduce this problem.

Brown [7] describes a recovery service that deals with operator errors in a mail server. Their system provides application-specific recovery that works well for a mail server, and while it is possible to extend the service to other applications, it is unclear how much effort is involved. In contrast, our system is geared towards server applications that do not necessarily have the clearly defined semantics of a mail server and hence our recovery techniques are more generic.

Sun [43] provides a safe execution environment (SEE) that enables users to try out new software (or configuration changes to existing software) without fear of damaging the system in any way. This is accomplished via a novel one-way isolation mechanism where processes running within the SEE are given read-access to the environment provided by the host OS, but their write operations do not affect the host until a commit point. The commit is performed if a consistency criteria is met or else the SEE is rolled back. This approach allows recovery only until the commit point and rollback may become more likely for long running SEEs.

Sandboxing techniques are complementary to our approach. They interposition code that allows blocking program actions that may compromise security, while recovery deals with intrusions after they occur. Janus [19] interpositions system calls using the `proc` file system. Systrace [29] notifies the user about system calls executed by an application. Then it generates a sandboxing policy based on user response. Sandboxing raises the issue of policy selection, i.e, determining what actions are permissible for a given piece of software.

10 Conclusions

This paper has presented Forensix, a robust, high-precision reconstruction and analysis tool for automated analysis and recovery of compromised systems. The salient features of Forensix are its kernel-level auditing of system activities, tamper-resistant logging on a separate back-end machine, and use of database technology to support efficient, high-level querying of data. A Linux-based implementation of Forensix was described, and a performance evaluation of it showed its overhead in terms of system throughput and storage capacity. While both costs are significant, they are

within the bounds of acceptability for many applications. Furthermore, technology trends, such as the rapid increase in disk capacity, will reduce these costs further in the future. The complete Forensix system is currently available at the project web site [1]

11 Acknowledgments

The authors would like to thank Mike Shea, Kenneth Po, Kamran Farhadi, Jin Choi, Sourabh Ahuja, Ho-Jeong An, Gary Yeung, Miria Grunick, Jennifer Johnson, and Jonathan Walpole for their contributions to the Forensix project.

References

[1] 4N6 Developers. The Forensix Project. `http://forensix.sourceforge.net/`.

[2] Armando Fox and David Patterson. Self-Repairing Computers. *Scientific American*, September 2004.

[3] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 259–272, 2004.

[4] M. Bernaschi, E. Gabrielli, and L. Mancini. REMUS: A Security-Enhanced Operating System. *ACM Transactions on Information and System Security*, 5(1), February 2002.

[5] M. Bishop and M. Dilger. Checking For Race Conditions in File Accesses. *Computer Systems*, 9(2):131–152, 1996.

[6] Bobbie
Harder. Microsoft windows system restore. http://msdn.microsoft.com/library/en-us/dnwxp/html/windowsxpsystemrestore.asp, April 2001.

[7] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Technical Conference*, pages 1–14, 2003.

[8] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centres. In *Proceedings of the Symposium on Operating Systems Principles*, pages 103–116, October 2001.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, pages 321–336, August 2004.

[10] I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[11] C. Cowan. Immunix: Adaptive System Survivability. http://www.immunix.org, http://www.cse.ogi.edu/sysl/projects/immunix, 1998.

[12] M. Crosbie and B. Kuperman. A Building Block Approach to Intrusion Detection. In *Recent Advances in Intrusion Detection (RAID 2001)*, Davis, California, October 2001. Springer.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage With CFS. In *Proceedings of the Symposium on Operating Systems Principles*, October 2001.

[14] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of OSDI*, December 2002.

[15] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. Technical report, 2000.

[16] Edward C. Bailey. *Maximum RPM*. Sams, August 1997.

[17] E. Eskin, W. Lee, and S. Stolfo. Modeling System Calls for Intrusion Detection with Dynamic Window Sizes. In *Proceedings of DARPA Information Survivability Converence and Exposition II*, June 2001.

[18] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[19] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, 1996.

[20] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[21] K. Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. Technical report, 1992.

[22] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[23] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles*, October 2003.

[24] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *ESORICS*, 2003.

[25] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Technical Conference*, pages 95–106. USENIX, January 1995.

[26] K. Mitnick. Takedown Transcripts: 1995 Feb 5 11:48:08. http://www.takedown.com/cgi-bin/transcript.pl?4002, February 1995.

[27] D. Pilania and T. cker Chiueh. Design, implementation, and evaluation of an intrusion resilient database system. Technical Report TR-124, SUNY, Stony Brook, April 2005.

[28] P. Porras. STAT: A State Transition Analysis Tool for Intrusion Detection. Technical report, 1992.

[29] N. Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 257–272, August 2003.

[30] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, 1998.

[31] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Technical Conference*, pages 183–195. USENIX, June 1994.

[32] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of Conference on File and Storage Technologies (FAST)*, January 2002.

[33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[34] sd and devik. Linux on-the-fly kernel patching without LKM. *Phrack*, (58), December 2001.

[35] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *IEEE Symposium on Security and Privacy*, May 2001.

[36] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, pages 63–78, August 1999.

[37] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *Proceedings of the ACM SIGCOMM*, pages 315–323, 1998.

[38] Slashdot. Slashdot FAQ. `http://slashdot.org/faq/`, 2000.

[39] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the USENIX Security Symposium*, pages 185–198, August 2000.

[40] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[41] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–180, 2000.

[42] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5):85–96, 2004.

[43] W. Sun, Z. Liang, R. Sekar, and V. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[44] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[45] The Honeynet Project and Research Alliance. Know Your Enemy: Phishing. `http://honeynet.evilcoder.org/papers/phishing/details/de-detailed.html`, May 2005.

[46] H. Xie and P. Biondi. Linux Intrusion Detection System (LIDS) Project. `http://www.lids.org/`.

[47] N. Zhu and T.-C. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.