Reconstructing System State for Intrusion Analysis

Ashvin Goel, Kamran Farhadi, Kenneth Po
University of Toronto

Wu-chang Feng
Portland State University

Abstract

The analysis of a compromised system is a time-consuming and error-prone task because commodity operating systems provide limited auditing facilities. We have been developing an operating-system level auditing system called Forensix that captures a high-resolution image of all system activities so that detailed analysis can be performed after an attack is detected. The challenge with this approach is that the large amount of audit data generated can overwhelm analysis tools. In this paper, we describe a technique that helps generate a time-line of the state of the system. This technique, based on preprocessing the audit log, simplifies the implementation of the analysis queries and enables running the analysis tools interactively on large data sets.

1 Introduction

When computer systems are compromised, system administrators face the task of determining answers to questions such as "where did the attack come from", "what vulnerability was exploited", and "which files did the attacker modify". This forensic analysis requires information about activities that occurred in the past on the system. Currently, this information is collected in a "lossy" manner. For example, system and application log files only track events based on what the system administrators or application developers think are necessary to log, and these files can be tampered with or deleted by the attacker. Vital information such as where an attacker connected from, and what happened afterward is not necessarily collected in these log files.

In recent years, several research efforts, such as ReVirt [2], FDR [17] and Forensix [6], have focused on capturing a high-resolution, tamper-resistant image of all system activities. ReVirt places a host system within a virtual machine and logs all non-deterministic events that can affect the system, allowing deterministic replay of the entire system at instruction granularity. FDR captures all interactions with the file system and Windows registry and uses this information for analyzing the behavior of systems, including software misconfiguration and security vulnerabilities, in production environments. Forensix intercepts all system calls and provides SQL tools that help with analysis of past system behavior. While these systems capture different types of events, they log these events at *all* times and they log *all* events of the given type. This *complete* log allows analysis of known intrusions as well as intrusions that become known in the future, since the log captures all system activities rather than just those that are considered important today.

The complete logging approach introduces performance and storage overheads, but these overheads have become manageable as computing, storage and networking costs have steadily decreased over time. For example, FDR requires 20 MB of storage per day with a storage-optimized log file format, while ReVirt and Forensix may need roughly 1 GB storage per day. Even with these higher requirements, a month of data (30 GB per machine) can easily be stored on today's disks. Furthermore, all these approaches have less than 1-10% performance overhead for typical server or desktop workloads.

While complete logging has become economically feasible, the amount of audit data generated can overwhelm simple data analysis techniques. For example, suppose an administrator suspects that an intrusion may have occurred and wishes to find all root-owned setuid files that existed on the system yesterday. Instead of trusting the host system, the administrator issues this analysis query on the audit log. This query requires scanning the audit log to determine when files were created, destroyed or their permissions or ownership was last changed. ReVirt would require replaying the system starting from a previously stored snapshot to recreate these events, while FDR and Forensix store these events. Even so, a root-owned setuid file that was created a long time back and whose attributes were never changed would also match the query and hence the entire audit log must be processed. This example shows that analysis can require scanning and processing of large amounts of audit data even for seemingly simple queries.

The problem with the raw audit log is that it consists of events, representing changes in system state, while analysis may require determining the state of the system at a given time or a time interval. For example, a system-call based logging system captures state-change events such as when a process is created or when a file or its attributes are modified, but the query described earlier requires determining the state of the system (in terms of root-owned setuid files) since yesterday. A simple method of reconstructing this state consists of sequentially processing all the state-change events, but the amount of data processing involved can slow the queries and limit their usefulness, especially since intrusion analysis is an inherently interactive process.

In this paper, we present a method for efficiently reconstructing the past states of a system. Our technique, implemented in the Forensix intrusion analysis system [7, 6], is based on storing the *lifetimes* of objects and attributes that are used in an analysis query. For example, we use a *file_owner* lifetime table to store information about the different owners

of a file over time. This lifetime table contains time intervals (start and end time) for each different owner of every file. With this table, it is straightforward to query for all root-owned files since yesterday since any such file is owned by root and has an end time greater than yesterday. Similarly, we use a *file_permission* lifetime table to store information about the different permissions of each file over time. With this table, it is simple to find out about setuid files that existed since yesterday. The results of our administrator's query would then consist of the common results obtained from the file_owner and the file_permission tables.

We have designed and implemented several lifetime interval tables in Forensix. These tables simplify the implementation of Forensix queries and greatly improve their performance. We use interval tables extensively in Forensix to implement tools that allow analysis of file accesses, tracking of persistent data, replaying process IO activities and tracking of dependencies among system objects such as sockets, files and processes [8]. We have applied these tools to analyze real attacks and our results show that our interval-table based analysis tools can be used interactively even when operating on large amounts of audit data.

The rest of the paper describes our approach in detail. Section 2 motivates the need for reconstruction of past system states and the complexity of designing the required analysis tools. Section 3 explains our state reconstruction technique, and Section 4 describes the analysis tools that we have implemented using our technique. Section 5 presents an evaluation of our approach, Section 6 describes the related work in the area, and Section 7 provides our conclusions.

2 Motivation

Over the last four years, we have been developing Forensix, an auditing and intrusion analysis system that monitors all process management, file system and networking-related system calls on a target host and logs these events into a MySQL database located on a secured "backend" system. Forensix provides analysis tools that are run entirely on the backend so that evidence is left intact on the target.

When implementing analysis queries on the logged events, we realized that there was a mismatch between the events and the queries. The events consist of changes in system state, while our analysis queries may require determining the state of the system at a given time or a time interval. For example, the Forensix database contains the `fork` and `wait` events that indicate the creation of a process and exit of a child process. Suppose we want to run a query that returns the names of processes that existed in the last hour. This query requires processing *all* the `fork` and `wait` events to determine the lifetimes of processes. Below, we describe several other scenarios that motivate the problem of analyzing system state from the raw audit events. Later, we will show how our approach solves the problems described below.

Scenario 1: Find files with owner=*O* and permission=*P* at time=*T*. An administrator suspects that someone has exploited a vulnerability to create an unauthorized setuid root binary and wishes to compare the setuid root binaries that currently exist on the system with those that existed a few days earlier. A general query of this type requires processing four different sets of events (file creation, change ownership, change permission and file deletion) that occur before time *T*.

Let S1 be the set of files owned by owner *O* at time *T*. This set is generated by using the file creation and change ownership events. These events help determine the *last* event associated with each file that occurred *before* time *T* and that set the owner to *O*. We also need to remove files from S1 that have been deleted before time *T*. Similarly, let S2 be the set of files that had permission *P* at time *T*. This set is generated by using the file creation, change permission and file deletion sets. The final result is obtained by intersecting sets S1 and S2. This relatively simple query is difficult to write using the raw events, and it is inconvenient because the user has to query various different types of events. Furthermore, the query is inefficient because all events of the four types must be examined even though only the *last* event before time *T* is relevant for any given file.

Scenario 2: Find the contents of directory=*D* at time=*T*. An administrator knows that the tarballs of a popular rootkit and a local-root exploit unpack into directories named rkid and xpl. He wishes to find all directories that may have had these names and retrieve the contents of these directories. This query requires processing all events that occur before time *T* and that create, rename or remove a directory entry from directory *D*. This query is inefficient because it requires processing or replaying all events related to directory *D* until time *T* to determine the contents of the directory.

Scenario 3: Find the path name of a file whose inode=*I* at time=*T*. An administrator suspects that someone has modified /etc/passwd and wishes to determine all accesses to the file and all names (hard links) and symbolic links associated with this file. The latter query needs to perform reverse name resolution from file identifiers (inode numbers) to path names. To do so, the file name of inode number *I* at time *T* must be determined by looking for the last event before time *T* that either created or updated a name for that inode. In addition, the inode number of the parent directory during that event must be known. This process of looking for the last event must then be performed recursively for the parent directory's inode number until the whole path is resolved. This query has to examine many different events and determine the relevant last events.

Scenario 4: Find processes whose effective user id=*E* between *Ts* and *Te*. An administrator is informed of a new exploit that allows the Apache user to run a setuid root binary and wishes to find all programs that ran with elevated privileges over the last two weeks. For this query, we need to consider the `fork`, `execve, setuid` and `wait` process management events. The first type of event can be used to find the set of processes that were created with euid set to *E*. The second type of event helps determine the set of processes that executed a setuid file whose owner was *E*, while the third type of event shows the set of processes that successfully changed their effective user id to *E*. The last type of event is used to

filter processes that exit before time *Ts*. This query is complicated because different processing is required for each set of events. Note that all the relevant events until time *Te* must be processed. For example, a process that is created much before time *Ts* with euid *E* and exits after *Ts* would match the query.

Scenarios 5: Find all processes whose lifetimes overlapped with the process whose name=*N*. During the analysis of an attack, an administrator finds that the wget program was run to download a "rk.jpg" binary. He wishes to find all server processes that were running at that time to confirm his hypothesis that the ftp daemon was attacked. This query requires determining the lifetimes of all processes, which requires processing *all* `fork` and `wait` events. In addition, we need to find the lifetimes of processes whose name is *N,* which also requires processing all `execve` events.

Scenario 6: Find root-owned setuid files that were executed by non-root processes. The administrator wishes to create a daily privilege escalation report. This query is, roughly speaking, a combination of the first and fourth queries and not described in more detail here. It has constraints on both file and process attributes, which makes it more complex to write than any of the previous queries.

3 System State Reconstruction

In the previous section we showed that state-based analysis queries can be hard to implement and may require processing of the entire audit log. This problem occurs because the audit log does not provide information directly about the value (or state) of an object or an object's attribute at a particular time or time interval. We preprocess the audit log to generate this information, thereby speeding up analysis queries and simplifying their implementation. In particular, we generate the *lifetimes* of objects and attributes that are used in queries. For example, the lifetime of a process is the time interval between when the process is created and destroyed. With this information, Query 5 shown in the previous section can easily determine processes that overlapped in time. Below, we describe our method in more detail.

3.1 Interval Tables

We derive the lifetimes of kernel objects and their attributes immediately after the Forensix audit log is loaded into the Forensix database. These lifetimes are stored in *interval* tables, and we refer to the process of creating these tables as reconstructing system state. We have identified several interval tables based on the requirements of our analysis tools. These tables are shown in Table 1. Each row of an interval table maps a system object such as a file, connection or process and optionally an attribute of this object to a lifetime, consisting of a start time *Ts* and an end time *Te*.

The `inode` interval table correlates a file identifier (inode number) to the lifetime of its names. In each row of this table, the start time is the time when the file name was initially created and the end time is when the file name was removed. For example, a new row is created in this table when a new file or a file name (a hard link) is created. The end time is updated when the file name is removed. A file rename is considered equivalent to a file name removal and a file name creation. In addition to the file name, this table contains the type of the inode (e.g., file, directory, symbolic link, device node, etc.) and the inode number of the parent directory. The `connection` interval table maps a connection to the lifetime of a connection. The `file_owner` and `file_permission` interval tables correlate a file with its owner and permissions so that each row represents a unique owner and permission for the file.

The `process` interval table correlates a process identifier with the lifetime of the process name. A process identifier with multiple names (`execve`) creates multiple entries in this table. The `process_owner` interval table maps a process identifier to the lifetime of the process owner (user and group id).

The main requirement for constructing these interval tables is that each system object should have a unique identifier over time. We used timestamps to create unique process identifiers (pid), file identifiers (inode number) and connection identifiers (connection_tuple). For processes, we used the process creation time. Files are uniquely identified with a device number, inode number and a generation number that is stored on disk by most commonly available Unix file systems today. The generation number is incremented when an inode number is reused. The connection tuple consists of source and destination addresses and ports. This tuple together with an inode associated with the connection uniquely identifies a connection over time. To speed up queries, we create database indexes on the unique identifiers in each interval table. Appendix A provides an example of how the interval tables are constructed from the raw events.

3.2 Queries with Interval Tables

The interval tables described above help simplify Forensix queries. Section 4 describes various intrusion analysis tools that we have developed using the interval tables. Here, we show how the queries described in Section 2 can be easily implemented with interval tables using SQL code. Readers unfamiliar with SQL can scan the rest of this section but should notice the simplicity of the code implementing these queries.

Query 1: Find files with owner=*O* and permission=*P* at time=*T*. The following simple SQL query provides file identifiers matching the query. The names of files can be derived from the file identifiers with Query 3 shown below. Note the use of time interval (`ts`, `te`) here and in all the queries below to determine system state.

```
SELECT f.inode
FROM file_owner f, file_permissions p
WHERE f.owner = O AND p.permission = P
  AND T BETWEEN (f.ts, f.te)
  AND T BETWEEN (p.ts, p.te)
```

Query 2: Find the contents of directory=*D* at time=*T*. This query, which lists the contents of a directory at a given time, takes advantage of the `parent_inode` information available in the `inode` interval table. It lists all file names that have the parent directory *D* at time *T*. If the directory is specified by

Interval table	Table columns	Events that update the table
inode table	inode+, file_name, parent_inode+, Ts, Te	create*, mkdir, link, symlink, mknod, rename, unlink, rmdir
connection table	inode+, connection_tuple+, Ts, Te	socketcall* (accept, connect, etc.)
file_owner table	inode+, owner, group, Ts, Te	create*, mkdir, symlink, mknod, chown*, unlink, rmdir
file_permission table	inode+, permission, Ts, Te	create*, mkdir, symlink, mknod, chmod*, unlink, rmdir
process table	pid+, inode+, file_name, parent_inode+, Ts, Te	fork*, execve, wait*
process_owner table	pid+, uid, euid, gid, egid, Ts, Te	fork*, execve, wait*, setuid*

For each interval table, the second column shows the columns of the interval table. The last column shows the events that update the interval table. The plus sign after inode, connection_tuple and pid shows that these system objects must be uniquely identified. The asterisk sign after certain events indicates that there are several variants of these events.

Table 1: Interval tables.

name, then the `inode` interval table can be used to first find the directory's inode number *D*.

```
SELECT i.file_name
FROM inode i
WHERE i.parent_inode = D
  AND T BETWEEN (i.ts, i.te)
```

Query 3: Find the path name of a file whose inode=*I* at time=*T*. This query requires a loop to find the path name one component at a time. The pseudo code is shown below.

```
var INODE = I
do:
  SELECT i.file_name, i.parent_inode
  FROM inode i
  WHERE i.inode = INODE
    AND T BETWEEN (i.ts, i.te)

  INODE = i.parent_inode
while INODE is not '/' # root inode
```

Query 4: Find processes whose effective user id=*E* between *T1* and *T2*. The following query operates on the `process_owner` interval table. The last two conditions search for overlapping time intervals.

```
SELECT p.pid
FROM process_owner o
WHERE o.euid = E
  AND T1 < o.te
  AND T2 > o.ts
```

Query 5: Find all processes whose lifetimes overlapped with the process whose name=*N*. This query is a little more involved and requires a temporal join of the `process` interval table with itself to find the overlapping intervals.

```
SELECT DISTINCT p2.pid
FROM process p1, process p2
WHERE p1.name = N
  AND p1.pid != p2.pid # ignore self
      # overlapping interval
  AND p2.ts <= p1.te
  AND p2.te >= p1.ts
```

Query 6: Find root-owned setuid files that were executed by non-root processes. This query is more complex than the previous queries because it has constraints on both file and process attributes. In addition, it requires data from the `execve` event. The `execve` event is stored in Forensix as a separate `exec` table. This table stores the event time stamp, the process id and the inode number of the file that was executed. The query below joins the `file_owner` and `file_permissions` interval tables (for root-owned setuid files), the `process_owner` interval table (for non-root processes) and the `exec` table to derive the query results.

```
SELECT e.inode
FROM file_owner f, file_permissions p
     process_owner_table o, exec e
WHERE f.owner = 'root'
  AND p.permissions has 'setuid'
      # non-root process
  AND o.euid != 'root'
  AND e.pid = o.pid
      # file that was executed
  AND e.inode = f.inode
  AND e.inode = p.inode
  AND e.time BETWEEN (f.ts, f.te)
  AND e.time BETWEEN (p.ts, p.te)
  AND e.time BETWEEN (o.ts, o.te)
```

4 Analysis Tools

The previous section shows that lifetime interval tables simplify the implementation of state-based queries. With the interval tables shown in Figure 1, we have built several powerful intrusion analysis tools including the *file-access* tracker, the *IO* tracker, the *directory* tracker and the *dependency* tracker, each of which presents a unique view of system state and changes to the state over time. We briefly describe these tools below. More details about the tools are available in our previous publications [6, 8].

The file-access tracker shows files that have been accessed or modified in a given time interval. This data can be voluminous so the tracker provides various filters (based on the type of events, file names and attributes, and process names and

attributes) that help limit the results. For example, Query 6 in Section 3.2, which determines root-owned setuid files that were executed by non-root processes, is a specialized case of the file-access tracker.

The IO tracker replays the IO performed by processes (process IO tracker) and reconstructs the contents of files (file IO tracker). The process IO tracker replays the writes of a process or process hierarchy. It can be used, for example, to replay the entire shell activity seen by a remote intruder. The file IO tracker allows recreating the contents of files at a given time. We use this tool for post-intrusion file-system recovery [8].

The directory tracker reconstructs the contents of directories at a given time. For example, the file-access tracker might show a directory that was created by an attacker. The directory tracker could show the contents of the directory after the attack even though the directory may have been removed by the attacker. The basic directory tracker is shown in Query 2 in Section 3.2.

The dependency tracker displays data dependencies between processes, files and sockets. For example, a dependency occurs when a process reads or writes from a file. A graph of such dependencies show the chains of events that led to an intrusion or are caused as a result of the intrusion [12]. We use this tool to generate such a dependency graph. The implementation uses the rows of the `process`, `inode` and `connection` interval tables as nodes in the dependency graph. The tracker implements several filters that help prune undesired edges from the graph for easy visibility. These filters use the attributes of the interval tables.

We plan to simplify an administrator's job by providing a library of prepackaged analysis queries that are run as part of a daily activity report or can be run using a web interface.

5 Evaluation

The previous sections have shown that the Forensix analysis queries are simpler to implement with interval tables. In this section, we show that our analysis tools run significantly faster with interval tables. Our experimental setup consists of a target machine and a backend machine both running AMD Athlon MP 2600+ CPU 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 uses the MySQL version 5 database. The target was run with the vulnerable services for approximately a week. During this time, the target was attacked once externally with the Wu-ftpd remote root exploit. Below, we present our analysis of the attack and the time it took to run the Forensix tools to analyze the attack.

5.1 Analysis of Ftpd Attack

Snort reported an anonymous FTP login on May 12 around 17:10 followed by command overflow attempts that contained shellcode. While Snort helps detect attacks, it provides little information about what actually happened on the system.

```
/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
```

Figure 1: File-access tracker output for ftpd attack.

To look for any recent changes in the file system, we ran the file-access tracker to list all the files or directories modified between 17:00 and 19:00 of that day. A partial report, shown in Figure 1, 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 the dependency tracker using the modified /usr/bin/killall (shown in Figure 1 by the file-access tracker) as one of the detection points. A partial resulting graph is shown in Figure 2. 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 `inode` interval table for any instance of a file creation within the /dev/pts directory 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 the `process_owner` interval table showed that the attacker's shell was run as root. Next we used our IO tracker tool to replay the shell. Key output from the shell session is shown in Figure 3.

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 via /dev/kmem and does not need a kernel with support for loadable kernel modules [14]. With the rootkit, the attacker tried to hide the fake crond process, but since we use LIDS [18] on the target system to disable writes to /dev/kmem, the attacker was not successful.

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 [16].

5.2 Analysis Results

The time taken to run each of the queries described above is shown in Table 2. These queries use the interval tables shown in Figure 1, and the table shows that these queries can be used interactively.

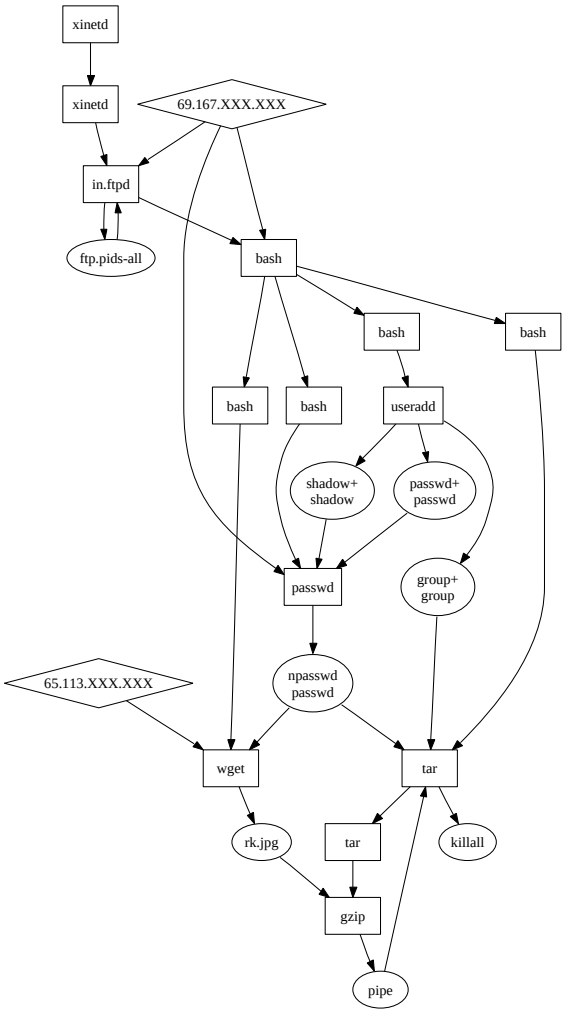

Figure 2: Tracking the FTP intrusion.

```
[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 =]
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
```

Figure 3: IO tracker output for the ftpd attack.

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 the children of the compromised in.ftpd process	1 s
Finding the listening port set by the attack code	< 1 s

Table 2: Time taken for each query.

Without the interval tables, analysis queries on the Forensix audit log are not only much harder to implement, but they essentially have to generate partial interval tables on the fly. We implemented the first two queries in Table 2 without using the interval tables (the other queries are much harder to implement without the interval tables). Their running times were 79 s and 33 s, which is a factor of 4-5 times slower. Our approach generates the interval tables once and hence queries can reuse the reconstructed state for faster analysis. When the target system is heavily loaded, the bottleneck in Forensix is the MySQL database loading time [6]. Interval table creation generally takes less than 10% of the loading time.

6 Related Work

Our intrusion analysis method is based on complete and tamper-proof logging. A similar approach is used in ReVirt [2] that places a system within a virtual machine and logs the VM-to-host interactions, allowing accurate system replay without requiring kernel integrity. ReVirt replays activity in linear time order. However, this approach is complementary to Forensix since it can be used to extract system call events during the first replay and then our backend system can be used for analysis. Garfinkel uses a similar VMM-based kernel introspection mechanism [5].

Similar to Forensix, FDR [17] logs interactions with persistent state and uses this information to analyze software misconfiguration. FDR focuses on using a customized storage format to minimize the amount of logging, which in turn also improves query performance. In contrast, Forensix uses the standard MySQL database but implements the interval tables to simplify the implementation of queries and improve their performance. We believe that it should be possible to combine these two approaches to derive the benefits of both.

System call traces have been used in the past to identify normal system behavior and then to automatically detect suspicious behavior or intrusions [9, 15, 3]. However, these approaches examine system-call patterns over a short window of 5-100 calls and are not designed to capture, store and analyze all system activity that has occurred in the past.

Tripwire [11] monitors the cryptographic hash and size of key system files and reports file accesses and modifications. Venema and Farmer developed the Coroner's Toolkit (TCT) [4] that uses file-system specific information for post-mortem analysis of a UNIX system. The Sleuth Kit [1] is a derivative of Coroner's Toolkit and provides file system in-

formation, file names and contents from file inode information and lists recently deleted files in a directory. Our analysis queries use the interval tables to provide this information without needing any knowledge of file system structure.

Our dependency tracker is directly motivated by the work on backtracking intrusions [12]. Complementary to our host-based intrusion analysis tools are network-based analysis tools such as SNORT [13] that capture and log network packets and help detect intrusions based on predefined rules that match packet headers or data. These analysis rules are filters that help reduce the amount of data that needs to be logged, but they only allow detecting known vulnerabilities.

7 Conclusions

Our intrusion analysis approach consists of capturing a complete system-level audit trail. The challenge is to provide tools that allow analyzing the large amount of audit data that can be generated. This data consists of changes in system state while analysis queries may require determining the state of a system at some time, such as just before or after an intrusion. We have shown that interval tables, containing the lifetimes of system objects or their attributes, ease the process of determining system state and allow implementing analysis tools that can be used interactively. With these interval tables, we have implemented several powerful intrusion analysis tools and used them to analyze real attacks. We have also been using the Forensix infrastructure to characterize system behavior with a view towards improving end-host security [10].

References

[1] Brian Carrier. The Sleuth kit & Autopsy. `http://www.sleuthkit.org/`.

[2] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2002.

[3] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[4] Dan Farmer and Wietse Venema. The Coroner's toolkit. `http://www.porcupine.org/forensics/tct.html`.

[5] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[6] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, and Jim Snow. Automatic high-performance reconstruction and recovery. *Journal of Computer Networks*, 51(5):1361–1377, April 2007. From Intrusion Detection to Self-Protection.

[7] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *Proceedings of the International Workshop on Security in Distributed Computing Systems (SDCS)*, June 2005. In conjunction with the International Conference on Distributed Computing Systems (ICDCS).

[8] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2005.

[9] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[10] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-level isolation and recovery with Solitude. In *Proceedings of the EuroSys conference*, April 2008. accepted for publication.

[11] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[13] Martin Roesch. Snort - Lightweight intrusion detection for networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 229–238, November 1999.

[14] sd and devik. Linux on-the-fly kernel patching without LKM. Phrack issue 58, December 2001.

[15] 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.

[16] The Honeynet Project & Research Alliance. Know your enemy: Phishing. `http://honeynet.org/papers/phishing`, May 2005.

[17] Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi-Min Wang, and Roussi Roussev. Flight data recorder: monitoring persistent-state interactions to improve systems management. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 117–130, 2006.

[18] Huagang Xie and et al. Linux intrusion detection system (LIDS) project. `http://www.lids.org/`.

A Implementing Interval Tables

We construct interval tables using a small number of SQL queries. For each table, at least two queries are needed, one for the start time and another for the end time for each row. The tables are updated whenever the audit log is loaded in the background into the Forensix database.

As an example, the SQL code shown below populates the `inode` interval table. The first query inserts new rows into the table and sets the start time for these entries. It searches the Forensix name_create_event table that stores all the events that create a new file name such as `creat`, `open`, `mkdir`, `link`, `rename`, `symlink` and `mknod`. The second and third queries update the end times of current rows in the `inode` interval table based on `unlink`, `rename` and `rmdir` calls available in the Forensix name_remove_event table.

```
# Insert rows for newly created files
INSERT IGNORE INTO inode
  ( inode, filename, parent_inode,
    begin_time )
SELECT e.inode, e.filename,
  e.parent_inode, e.time
FROM name_create_event e
WHERE e.returncode >= 0

# update end times for existing rows
CREATE TEMPORARY TABLE temp_inode
FROM inode_table i, name_remove_event e
WHERE e.returncode >= 0
  AND i.parent_inode = e.parent_inode
  AND i.filename = e.filename
  AND i.end_time is not set
  AND e.time > i.start_time
GROUP BY i.id;

UPDATE inode i, temp_inode t
SET i.end_time = t.end_time
WHERE i.id = t.id;
```