INTRUSION ANALYSIS AND RECOVERY

by

Kamran Farhadi

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Intrusion Analysis and Recovery

Kamran Farhadi

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

When intrusions occur, two of the most costly, time-consuming, and human-intensive tasks are the analysis and recovery of the compromised system. This thesis uses a complete log of all system activities for post-facto analysis and recovery, and it shows how historical analysis tools can be implemented easily and efficiently over this complete log. These tools allow detailed analysis of real attacks.

This thesis also describes a framework for efficiently recovering file-system data after an intrusion occurs or after some damage is caused by system management error. Our approach uses an efficient redo recovery approach and ensures that no legitimate data is lost after recovery by using automated *conflict resolution* algorithms to isolate compromised objects that are needed by legitimate operations. This framework is fully implemented and a detailed evaluation shows that it can correctly recover file-system data from a wide range of incidents.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Ashvin Goel. The list of reasons I need to thank him is longer than this thesis, but in particular I would like to thank him for his friendship, guidance and high standards in research which taught me a lot.

I am thankful to graduate students Kai Yi Po, Zheng Li and Thomas Liu at the University of Toronto who helped me with my research and most importantly provided their friendship. I would also like to thank Professor Eyal de Lara for his valuable input and feedback on my research.

Finally, I would like to thank the University of Toronto as well as the Department of Electrical and Computer Engineering for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rapid increase in the number of security incidents reported in the last 15 years [5] and the constant evolution of threats has led to development of several defenses against intrusions. However, there are strong reasons to believe that even with these defense mechanisms, intrusions will still occur. For example, the best defense against intrusions, implementing fully secure and vulnerability-free applications, is not practical because of the unknown nature of new vulnerabilities and the amount of time, code and expense required to secure all existing and new applications. The next line of defense is the use of intrusion detection systems (IDS) that apply a variety of techniques to identify attack signatures or anomalous behaviour in a system. Unfortunately, IDSs are not ideal and can miss detection of some intrusions.

When an intrusion occurs, the system administrator needs to analyze it to understand the nature of the vulnerability that was exploited and the extent of damage that was caused. This post-facto intrusion analysis process is time intensive and highly error prone, because it is performed manually or with the help of rudimentary and hard-to-use tools. The Coroner's Toolkit [9] and the Sleuth Kit [4] are classic examples of intrusion analysis tools that sift through compromised systems to gather information such as the list of current processes or connections, application- or system-level log files, and unallocated blocks which contain deleted files. These tools are insufficient for detailed intrusion analysis because the available information about past activ-

ity is incomplete and unstructured. For example, system log files are "lossy" and only track events based on what the system administrators think are necessary to log. Vital information about where a hacker connected from, how the hacker entered and what the hacker did after he entered is not necessarily collected in the log files, or these files may have been tampered or deleted by the hacker.

This thesis focuses on developing comprehensive, efficient and easy-to-write *intrusion analysis* tools. Intrusion analysis seeks to answer questions such as "where did the attack come from", "what vulnerability was exploited", and "what has been damaged or which files did the attacker modify". Our analysis approach consists of two components, complete auditing and state reconstruction. We gather an accurate, high-resolution log of all system activities. In particular, we use the Forensix auditing system to securely and accurately log all operations related to files, processes and sockets [12]. This *complete* audit 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.

With the complete audit log, we designed and implemented a novel reconstruction technique that simplifies the implementation of historical intrusion analysis queries and allows running these queries near real-time on large data sets. Our reconstruction approach consists of storing the lifetimes of different objects and their attributes. Consider an analysis query that requires finding all files owned by a malevolent user at a given time. For this historical query, we create an *owner* lifetime interval table to store information about the different owners of each file over time. Each row of this table contains the time interval (start and end time) during which a certain file had a certain owner. With this table, which can be pre-generated, it is straightforward to find all files with a certain owner at a given time since all such files should have a row in the owner interval table with that owner and an interval which contains the specified time. We use this technique to implement various types of post-facto intrusion analysis tools as well as implement a recovery framework described below.

The second focus of this thesis is *intrusion recovery* of compromised systems. Intrusion re-

covery typically involves many manual steps: installation of a new system image that includes the operating system and all applications, installation of software patches that fix known vulnerabilities, and retrieval of uncorrupted user data. Today, snapshot-based file-systems [28, 34] provide a well understood and commonly deployed recovery solution [39]. This method gets rid of all corrupted data, but unfortunately, it also gets rid of useful data not related to the intrusion, and then this data must be manually retrieved or recovered separately.

The goal of intrusion recovery is to preserve all legitimate data while reverting the effects of attack-related (or tainted) file-system modification operations. Our recovery approach is based on the separation of attack-related activities from other legitimate activities. This separation is performed with an existing taint analysis method that uses the audit information to determine tainted system objects and operations [13]. Then, we define a novel framework that recovers only those parts of the file system that were affected by the attack and preserves the effects of all legitimate activities.

The recovery framework reverts the effects of tainted operations by selectively replaying legitimate operations on tainted file-system objects. However, legitimate operations that need to be preserved may depend on tainted operations. For example, a legitimate file may have been created in a tainted directory and simply removing the tainted directory conflicts with the legitimate file that needs to be preserved. To ensure that legitimate operations are not reverted, this thesis defines *conflict resolution* algorithms to isolate tainted operations. To do so, file-system operations are separated into name, content and attribute operations. This approach simplifies resolution, allows recovery actions that are suited for each type of operation, and enables fully automatic name and attribute conflict resolution.

1.1 Challenges

In general, intrusion analysis and recovery raises three types of challenges. First, we need to log all system activities. With the rapid and continuous decline in computing, networking, and

storage costs, this type of logging is now technically and economically feasible [35, 7, 19, 12].

The second challenge is that the audit log should allow efficient analysis queries and easy-to-write analysis tools. The raw audit log raises two problems: 1) the large amount of log generated can overwhelm traditional data analysis techniques and, 2) the raw audit log consists of changes in system state, such as when a process is created or when a file name or attribute is modified, while analysis often requires determining the state of the system at a given time or a time interval such as just before or after an attack. To do so, one needs to reconstruct the system state from the "state change" audit log. A simple method for recreating state consists of sequentially processing all the log. Unfortunately, the large amount of data processing involved can slow the queries which limits their usefulness since intrusion analysis is an inherently interactive process. In addition, as shown later, implementing analysis queries with this approach requires non-trivial effort.

Finally, with intrusion recovery, the challenge is to identify attack operations and revert them only. The recovery process should be able to efficiently restore each tainted file-system object to a clean state without losing legitimate data or violating file-system consistency. In addition, legitimate objects or operations may depend on tainted objects. In this case, resolution methods need to be defined so that legitimate objects can be preserved.

1.2 Contributions

This thesis shows that comprehensive, efficient and easy-to-write intrusion analysis tools can be implemented using *lifetime intervals* (or interval tables) of different attributes of system objects. Interval tables provide a powerful mechanism for recreating the historical state of a system from the raw audit log. We define and implement several interval tables based on the requirements of analysis tools and show how these interval tables simplify the implementation and improve the performance of queries in the Forensix auditing system [12]. We use the interval tables to implement a range of host-based analysis tools including, file-access tracker,

file-contents constructor and shell activities tracker. In essence, we show that using a complete audit trail, it is possible to subsume several existing intrusion analysis tools. To prove the utility of these tools, we apply them to real attacks, describe analysis results and show how one can query and analyze large audit logs interactively.

As the second contribution, this thesis describes a framework for recovering file-system data after an intrusion occurs or after some damage is caused by system management error. We use an existing taint analysis method [13] to identify attack-related activities. Our approach reverts the effects of the activities and ensures that no legitimate data is lost after recovery. We provide automated conflict resolution algorithms to isolate tainted objects which cannot be reverted because they are needed by legitimate operations. This framework is fully implemented and a detailed evaluation shows that it can correctly recover file-system data from a wide range of intrusions as well as erroneous system management activities.

1.3 Thesis Structure

Chapter 2 briefly describes the Forensix auditing system used in this research. Chapter 3 defines the infrastructure or lifetime intervals (interval tables) for intrusion analysis and Chapter 4 presents a number of tools on that basis. Chapter 5 defines the framework for intrusion recovery and provides algorithms to resolve conflicts between tainted and legitimate file-system operations. Chapter 6 evaluates the intrusion analysis tools and recovery algorithms in detail. Chapter 7 presents the related work in intrusion analysis and intrusion recovery areas. Finally, chapter 8 presents the conclusions and directions for future work.

Chapter 2

Forensix Auditing System

For intrusion analysis and recovery, we need to track the operations of three types of kernel objects, processes, files and socket connections. To do so, this thesis uses the Forensix system [12] that audits all kernel operations related to process management, file system and networking. In particular, Forensix monitors and logs all the relevant system calls and all their arguments.

Figure 2.1 shows the Forensix architecture. The target system, which provides services to the public network, is potentially vulnerable. The Forensix kernel logger audits all process management, file system and networking system calls on the target system and transmits the audit log over a dedicated network to a secured backend system where the log is stored in append-only files. Separating the backend from the target machine helps to ensure that logged information cannot be destroyed easily. In the backend system, the audit log is batch-loaded into a MySQL database periodically or on demand. Also, right before Forensix is started, a file-system snapshot of the target system is taken manually and stored on the backend machine. The audit database and the file-system snapshot are then used by our analysis and recovery tools which operate entirely on the backend and leave evidence intact on the target.

We assume that the logging system on the target is not corrupted as a result of an attack. Since Forensix runs in the kernel on the target system, this implies the assumption that the

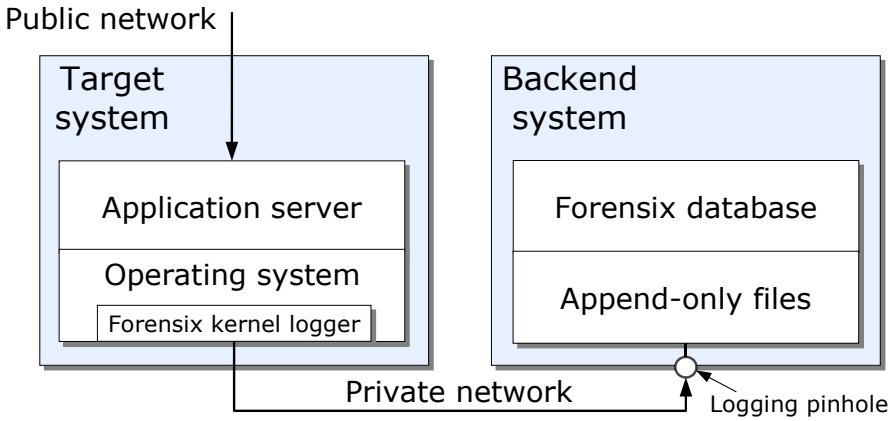

Figure 2.1: The Forensix architecture

applications but not the kernel on the target can be vulnerable. To reduce the risk of kernel intrusions, Forensix uses the Linux Intrusion Detection System (LIDS) [41], a kernel patch that adds Mandatory Access Control (MAC) and other security enhancements to the Linux kernel. Forensix uses LIDS particularly to disable 1) user-level writes to kernel memory, 2) user-level writes via the raw disk interface, 3) writing to the kernel or Forensix binary files, and 4) the loading of kernel modules. These simple measures make current root-kits ineffective [11, 29].

Forensix uses the Linux Security Modules facility [40] to capture information that helps to unambiguously determine the identity of system objects such as files, sockets and processes when they are accessed during system calls. This approach provides accurate ordering of events and race-free auditing [10]. Although Forensix uses kernel-based logging, it could, in principle, use other auditing mechanisms such as VM-based auditing that can provide additional resistance to attacks on the logging mechanism [7, 11].

To facilitate log data analysis, each kernel object of interest (sockets, processes and files) must be assigned an identifier that remains unique over time. For sockets and processes, the Forensix kernel logger attaches a creation time-stamp to the socket and the process id. To track operations on a file object, Forensix uses an object identifier, which for Unix-based files is the inode number. However, since inode numbers can be reused after an object is removed and these numbers are not unique across devices, we should use the three tuple (device, inode,

generation number) to uniquely identify file-system objects. This tuple is called the object id. Note that the generation number is used to differentiate between an inode before and after it has been deleted and reused.

The audit log captured from the target system is stored in the backend Forensix database. This database consists of several tables each of which stores similar types of operations on kernel objects. For example, all operations which modify the contents of files are stored in one table, while the name of files that are executed are stored in another table. This separation of operations allows efficient and easy access to the logged data since query-based accesses to each table can be independently optimized using table-specific indexes.

Chapter 3

Intrusion Analysis Infrastructure

The Forensix system provides the auditing infrastructure for intrusion analysis. Our goal is to use the audit data to develop efficient and easy-to-write intrusion analysis tools. Unfortunately, the large amount of raw audit data generated by Forensix can overwhelm traditional data analysis techniques. This chapter first motivates the problem associated with implementing historical intrusion analysis queries and then describes a framework that allows speeding up and simplifying the implementation of these queries. With this framework, we implement several post-facto intrusion analysis tools (described in Chapter 4) and intrusion recovery tools (described in Chapter 5).

3.1 Motivation

Raw audit data typically consists of changes in system state. For example, with Forensix, the `fork` and `wait` events indicate the creation of a process and exit of a child process. Intrusion analysis and recovery queries, on the other hand, require determining the state of the system at a given time or a time interval. For example, one may wish to know the names of processes that existed in the last hour. This query requires processing *all* the `fork` and `wait` audit events to determine the lifetimes of processes. Below, several other scenarios are described to motivate the problem of analyzing system state from raw audit data. These scenarios are chosen based

on our experiences with building intrusion analysis tools that are described later in Chapter 4.

Scenario 1: Find files with owner=*O* and permission=*P* at time=*T*. Suspecting that someone has used a ptrace execve race to create an unauthorized setuid root binary, an administrator 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 that occur before time *T*. These events are file creation (`mkdir`, `mknod`, `create`, `symlink`), change ownership (`chown*`[1]), change permission (`chmod*`) and file deletion (`rmdir`, `unlink`).

To find files that were owned by *O* at time *T*, we need to use the first two sets (file creation and change ownership) and determine for each file[2] the *last* event that occurred *before* time *T* and that set the owner to *O*. In addition, we need to remove files that have been deleted before time *T*. Similarly, to find files that had permission *P* at time *T*, we need to use the file creation, change permission and file deletion sets. The final result is obtained by intersecting the two sets (the *and* condition). This relatively simple query is difficult to write using the raw data 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*. Knowing that popular rootkit and local root exploit tarballs unpack into directories named rkid and xpl, an administrator wishes to find all directories that ever had these names as well as the contents of these directories. The latter query requires processing all events that occur before time *T* and that create an entry (`mkdir`, `mknod`, `create`, `symlink`, `link`), rename an entry (`rename`) or remove an entry (`rmdir`, `unlink`) from directory *D*. This query is inefficient because it requires processing

[1]The asterisk after an event indicates that the event has multiple variants. For example, `chown` and `fchown` perform similar operations.

[2]The queries described here work on file identifiers or inode numbers. Later, we describe queries that convert inode numbers to file names.

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*. Suspecting that someone has modified /etc/passwd, an administrator wishes to find all accesses to the corresponding inode as well as all names (hard links) and symbolic links associated with this file. The latter query performs reverse name resolution from file identifiers (inode numbers) to path names. To do so, first 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 (`mkdir`, `mknod`, `create`, `symlink`, `link`) or updated (`rename`) 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 last events that are relevant.

Scenario 4: Find processes whose effective user id=*E* between *Ts* and *Te*. Having been informed of a new exploit that allows the apache user to run a setuid root binary, an administrator wishes to find all such privilege escalations over the last two weeks. For this query, we need to consider the `fork*`, `execve`, `setuid*` and `wait*` 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 down-

load 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.

Above examples show that analysis using the raw audit data is challenging, and most of the times it requires processing large amounts of data.

3.2 System State Reconstruction

The previous section showed that analysis and recovery queries often require determining the historical state of a system. While this state is not directly available from the raw audit data, it can be easily derived once the *lifetimes* of different objects or their attributes is known. For example, in Query 5 in the previous section, overlapping processes can be easily found once the lifetimes of all processes is known. Knowing these lifetimes can simplify as well as speed up the queries significantly.

We create this lifetime information by pre-processing the Forensix audit log and store this information in auxiliary tables that are called *interval* tables. These tables store the lifetimes of objects or their attributes, and we refer to the process of creating these tables as reconstruction of system state. We will show later that while reconstructing this state involves some initial cost, it enables running queries efficiently and it simplifies the implementation of these queries.

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, permission, Ts, Te	create*, mkdir, symlink, mknod, chown*, 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 3.1: Interval tables

3.2.1 Interval Tables

We have identified several useful interval tables based on the requirements of our analysis tools. These tables are shown in Table 3.1. The data in these tables is obtained from the Forensix event data. Each row of an interval table maps a system object such as a file or connection or process and (optionally) an attribute of this object to a lifetime, which consists 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. For each row in this table, the start time is the time when the file name was initially created and, similarly, 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 (`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` interval table correlates a file with its owner, group and permissions so that each row represents a unique owner, group 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 interval tables is that each system object should have a unique identifier over time. For the tables in Table 3.1, we need to create unique process identifiers (pid), file identifiers (inode number) and connection identifiers (connection_tuple). To disambiguate processes, the kernel-level event logging subsystem shown in Figure 2.1 appends a process creation time-stamp to each pid. 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 the connection inode (used to determine reads and writes to a connection) uniquely identifies a connection over time. To speed up queries, we create database indexes on the unique identifiers in each interval table.

Section 3.3, provides a detailed example of how the interval tables are constructed from the raw data.

3.2.2 Queries with Interval Tables

The interval tables described above greatly simplify analysis queries written for the Forensix system. Below, we show how the queries described in Section 3.1 can be easily implemented with the interval tables. These queries are used as building blocks for implementing the comprehensive set of intrusion analysis tools that are described later in Chapter 4.

The queries in our system are implemented using SQL code. Readers unfamiliar with SQL can scan the rest of this section but should notice the simplicity of the code implementing these complex queries.

Query 1: Find files with owner=*O* and permission=*P* at time=*T*. The following simple SQL query provides the results for this query. The names of files can be derived from the inode numbers returned by this query using Query 3 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
WHERE f.owner = O AND f.permission = P
  AND T BETWEEN (f.ts, f.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 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.

```
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 simple query operates on the `process_owner` interval table. Note that the last two conditions search for overlapping time intervals.

```
SELECT p.pid
FROM process_owner p
WHERE p.euid = E
  AND T1 < p.te
  AND T2 > p.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 about the `execve` event. The `execve` data 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 data from the `file_owner` interval table (for 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,
     process_owner_table p, exec e
WHERE f.owner = 'root'
  AND f.permissions has 'setuid'
      # non-root process
  AND p.euid != 'root'
  AND e.pid = p.pid
      # file that was executed
  AND e.inode = f.inode
  AND e.time BETWEEN (f.ts, f.te)
  AND e.time BETWEEN (p.ts, p.te)
```

3.3 Implementation of Interval Tables

In this section, we describe the implementation of the interval tables which are used by our analysis tools. The tools are described in Chapter 4. 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 of an entry. The tables are updated whenever the audit log is loaded in the background into the Forensix database.

As an example, Figure 3.1 shows the basic SQL code that populates the `inode` interval table. Recall that each row of this table contains the time when a file name was initially created (start time) and the time when the file name was removed (end time). The first query inserts new entries 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 entries in the `inode` interval table based on the `unlink`, `rename` and `rmdir` calls available in the Forensix `name_remove_event` table (note that the `rename` event consists of a name creation as well as name removal operation). To find the correct end time, we use a GROUP BY clause to match the creation of each file name with the earliest removal of that name in the same directory since other files with the same name may be created and removed in that directory at other times.

The object id (inode number), by itself, is not sufficient for tracking the names of an object since file objects can have multiple names. To track names, the `inode` interval table maintains a name id (not shown in the figure 3.1) for each name of the file that consists of the tuple (object id, creation id). When a file object is created, it is assigned a starting creation id, and when a new name for the file is created (e.g., with the `link` system call), this new name is given a new creation id. The creation id does not change when a name is updated (e.g, with the `rename` system call) while the removal of a name ends the lifetime of the name id associated with that name. Our name id approach allows tracking name operations independently of file object (content, attribute) operations.

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

# Find end times for existing rows
CREATE TEMPORARY TABLE temp_end_time
SELECT i.id, min(e.time) AS end_time
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 end times
UPDATE inode i, temp_end_time t
SET i.end_time = t.end_time
WHERE i.id = t.id;
```

The first query creates new rows and sets the start time of these rows in the `inode` interval table. The second and third queries find and update the end times of existing rows.

Figure 3.1: SQL code for implementing the `inode` interval table.

Other interval tables are implemented using queries that are similar to the `inode` interval table. Table 3.1 shows the set of events that are queried to update each interval table.

Chapter 4

Intrusion Analysis Tools

The previous chapter presented our approach of using interval tables to reconstruct historical system state. This chapter describes how these tables can be used to build powerful and efficient analysis tools that subsume many existing host-based intrusion analysis tools. A detailed comparison with related approaches will be presented in the chapter on related work (Chapter 7).

Below, we present four types of tools that allow analysis of intrusions that occur via system call operations. In particular, these tools examine the operations performed by three types of kernel objects, processes, files and network sockets, and the interactions between these objects. The *directory tracker* lists the contents of directories, the *file-contents constructor* recreates the contents of files, the *file-access tracker* shows files that have been accessed or modified by a process or set of processes in a given time interval, and the *shell-IO tracker* replays the IO performed by shell processes. The first two tools help analysis of operations on files (and directories) while the latter two tools cover interactions between processes, files and sockets. For example, the shell-IO tracker allows replaying the IO performed by a process over a character device file or a socket.

Our evaluation in Chapter 6 shows that these tools are quite comprehensive and help analysis of real intrusions. While it is possible to implement other types of tools using the Forensix

audit data, the goal of this thesis is to show that the reconstruction of system state via interval tables provides a methodology for creating historical analysis tools such as the ones we have developed.

4.1 Directory Tracker

The directory tracker lists the contents of a directory at a given time. For example, the file-access tracker (described next) 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 later.

The basic directory tracker is shown below. It lists the contents of a directory with the object id D at the given time T. The simplicity of the query results from using the `inode` and the `file_owner` interval tables. The `inode` table provides the contents of the directory while the `file_owner` table provides the meta-data information such as the owner of the files.

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

The input to the directory tracker is either an object id or a path name, and the time at which the object/path name is to be analyzed. If the input is a path name, then we convert this path name to an object id as described below and then use the object id in the query above.

Resolving path name to Object Id

A path name in the Unix file-system is hierarchical and starts with the root directory "/", then it may contain several components each of which can be a directory, a symbolic link, or may

have a special meaning such as "..". Therefore, to obtain the object id of a path name at a given time, we start from the root node and traverse and resolve each path component to an object id until we reach the end of path name. The query below finds the object id of the path component N which is in directory D (at the given time T). This query is performed repeatedly to derive the object id of a path name.

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

Note that symbolic links in the path name are resolved to their target string. Also, special meaning path components are resolved so that the path name is canonicalized at the end.

4.2 File-Contents Constructor

The file-content constructor allows recreating the contents of files at a given time. To do so, it first derives the object id of the given file's path name using the algorithm in section 4.1. Then it replays all events before the given time which modified the contents of the file with the object id. It uses the following simple query which retrieves data from the `write` table in Forensix that stores the inode number, data, position and length of writes for `write` events and displays it in time order.

```
SELECT data, position, length
FROM write
WHERE inode = I
ORDER BY time;
```

Files which exist before Forensix was started are obtained from a file-system snapshot taken right before Forensix is started. This tool is used during file-system recovery (see Chapter 5) because it is able to recreate any version of a given file at any given time.

One optimization to this tool is to replay only those file events after the last complete truncation of the file. This simple optimization proves to be effective for several files that are modified in non-appending mode and usually are completely truncated before their modified content is written back to disk.

At this time, the main limitation of file contents constructor is that it does not recreate files that have memory mapped writes. However, Snow et al [32] are currently working on a project to modify Forensix to track memory-mapped files via page cache auditing.

4.3 File-Access Tracker

The file-access tracker in its simplest form displays the access or the modification times of files. To do so, it considers all events that read, execute, create, modify or remove the contents or the attributes of files. This data can be voluminous so the tracker provides several different types of filters that limit the results. These filters can provide additional information also. These filters are briefly described below.

Event type: This filter limits results based on the type of access. For example, it can show only create events.

Time: Limits results by time interval.

Last access: This filter only shows the last access or modification time of a file.

File names: Used to filter results based on names of files as well as show the names of files. For example, one may only be interested in files modified in the `/bin` directory. This filter uses the `inode` interval table.

File attributes: Used to filter results based on file attributes such as type of file (file, directory, symlink, etc.), owner, group or permissions. For example, one may only be interested in root-owned files. This filter uses the `file_owner` interval table.

Process names: Filter results based on accesses performed by certain types of processes. This filter can be either the process name or the process executable. This filter uses the `process` interval table.

Process attributes: Filter results based on process attributes such as uid, euid, etc. This filter uses the `process_owner` interval table.

The file-access tracker also allows grouping by any of the attributes above. For example, one can view the frequency of file accesses in directories modified by a server program such as the Apache web server and group this information both by directories and on a daily basis. Since the behavior of server programs is relatively well characterized [22], such a view would quickly show if a new directory was modified or if the directory access patterns had changed considerably on a given day compared to previous days. The user could then look for more detailed views for that day.

The implementation of these filters involves a database join between the interval tables and the underlying Forensix tables. The Forensix tables contain the pid and the inode number for the file access events. The interval tables contain either the pid or the inode number that is used as the join condition. For example, Query 6 in Section 3.2.2, which determines root-owned setuid files that were executed by non-root processes, is a specialized case of the file-access tracker. In that query, the `file_owner` and the `process_owner` interval tables are joined with the auxiliary `exec` table that contains the pid and the inode number of the file that was executed. The join operation allows combining all or some of these filters to form powerful queries.

The output of the filter are file object ids that need to be resolved to full path names. This resolution is described below. After this resolution, the result is filtered based on path name conditions, if there exist any such conditions.

Resolving Object Id to path name

Resolving an object id to a path name is similar but simpler to resolving a path name to an object id. The difference is that the path name is created in reverse order. Using the query below, the object id I is resolved to a name and parent directory at time T and the process is repeated for the parent directory until we reach root ("/"). At this point, all the path components are obtained and the full canonicalized path name can be created.

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

4.4 Shell-IO Tracker

The shell IO tracker replays the user IO performed (what the user typed and what the user saw) in an interactive shell login process. The input to this tool is the pid of the process which started a login shell. Processes of this type can be easily identified since they typically execute a shell program such as /bin/sh and open a character device file such as /dev/pts/0 to send shell output and to echo back a user's typed character. To replay a user's shell, first the process tree of the main shell process should be generated so that the whole session is captured. Children of a process PID can be found by the query below which is executed recursively until the whole process tree is generated.

```
INSERT INTO tmp_pid
SELECT child_pid
FROM fork
WHERE pid = PID
```

Then, all the writes issued by any of the processes in `tmp_pid` to the shell's character device file (the device path name is resolved to object id I) are retrieved using the query below. We also retrieve the time of these writes so that we can replay the shell output in time order and at

the same speed as the shell user's interaction. This query replays all activity seen by a local or remote intruder.

```
SELECT w.data, w.time
FROM write w, tmp_pid p
WHERE w.pid = p.pid
  AND w.inode = I
ORDER BY w.time
```

The basic implementation of the shell-IO tracker applies to terminal emulators such as xterm and remote-login programs such as sshd both of which use pseudo-terminal master and slave files (ptmx and pts) to implement interactive shells. Other types of shells are slightly harder to trace. For example, many remote shells and backdoors only use sockets to interact with user and do not create a terminal or a pseudo-terminal on the local machine. Typically, at the beginning, these shells duplicate the socket's file descriptor three times with file descriptors 0, 1 and 2 which represent `stdout`, `stdin` and `stderr` respectively. The query below uses this heuristic to identify the main shell processes and the inode of their communicating socket. Shell IO can then be retrieved using the socket inode as the input to the same code described above.

```
SELECT pid, inode
FROM dup_event
WHERE event = dup2
  AND inode_type = socket
  AND newfd IN (0,1,2)
```

Chapter 5

Intrusion Recovery

The second goal of this thesis is file-system recovery after an intrusion. Our recovery approach reverts the effects of attack-related (tainted) file-system modification operations while preserving all legitimate data. This approach has been implemented as part of the Taser intrusion recovery system [13]. Below, we provide an overview of Taser. Then we describe our file-system recovery model and the recovery algorithm. As part of recovery, legitimate operations may sometimes depend on tainted operations. To preserve the effects of such legitimate operations, we define conflict resolution procedures that isolate the tainted operations rather than reverting them. Our recovery algorithms use the intrusion analysis infrastructure and some of the analysis tools described in Chapters 3 and 4.

5.1 Overview of Taser

The Taser system recovers file-system data after an intrusion or management error by reverting the file-system modification operations affected by a system compromise or a system management error while preserving the modifications made by legitimate processes. From now on, we use the term intrusion to mean a system compromise as well as a management error.

The Taser architecture consists of three main components: Auditor, Analyzer, and Resolver. The Auditor consists of the Forensix auditing system described in Chapter 2, and it runs in the

background during normal system operation and creates an audit log of all system activities including file-system operations. The Analyzer and Resolver are executed by an administrator during the recovery process. Recovery is started after an intrusion has been detected externally such as by an intrusion detection system (IDS) or by an administrator. The Analyzer uses the audit log to determine the set of *tainted* file-system objects that were affected by the intrusion. The Resolver uses this set of tainted objects and the audit log to revert file-system modifications resulting from the intrusion. To revert operations, the Resolver *selectively* replays legitimate file-system operations on the tainted objects. Below, we provide an overview of the Analyzer and the Resolver.

The Analyzer component of Taser determines the set of tainted file-system objects by creating dependencies between sockets, processes and files based on entries in the audit log. Socket connections form initiating points for remote attacks, processes issue operations that create other dependent processes or files, and file accesses cause additional dependencies, and, in addition, files are the persistent state of the system that need to be recovered.

A dependency is caused when information flows from one kernel object to another via a system-call operation. For example, when a process writes to a file, the file becomes dependent on the process. Similarly, a process becomes dependent on a file when it reads the file. Table 5.1 shows the dependency rules between the kernel objects that are considered by the Analyzer. These rules are used to taint a dependent object when the source object is tainted. Each dependency, which always involves a process, is caused by the type of operations shown in the corresponding row. The last column of Table 5.1 shows some of the key system call operations that constitute each type of operation. For the details of each dependency rule, the reader is referred to a more complete description of the Analyzer [13].

The Analyzer's tainting algorithm derives the set of tainted objects using the Forensix audit log, the dependency rules shown in Table 5.1 and an initial set of tainted objects, known as detection points, that are provided by an intrusion detection system (IDS) or an administrator. Detection points can either be the source of an attack (e.g., a malicious socket connection that

Dependency Rule	Type of Operation	Operation
Process → Process	Fork	fork, vfork
	IPC, Signals	pipe, kill, mmap
Process → File	Write file content	creat, truncate, unlink, write
	Write file name	creat, link, symlink, rename, unlink
	Write file attributes	creat, unlink, chown, chmod
File → Process	Execute	execve
	Read file content	read
	Read file name	open, truncate, chown, chmod
	Read file attributes	open, truncate, chown, chmod
Process → Socket	Write	write, socketcall, sendfile
Socket → Process	Read	read, socketcall

Table 5.1: Dependency rules between processes, files and sockets

originates an intrusion), or the result of an attack (e.g., some strange files identified by a host IDS). When the detection points are not the source of an attack, the algorithm goes into an initial tracing phase that starts from the detection points and scans the audit log backwards to trace the source objects of the attack. The algorithm then switches to the propagation phase that starts from the source objects of the attack and scans the audit log forwards and taints objects and operations affected by the intrusion.

This tainting algorithm is conservative and taints all attack-related objects but it can also result in a large number of false dependencies leading to legitimate objects being marked tainted. For example, suppose an attacker appends a malevolent account to the /etc/passwd file. The tainting algorithm will taint this file and all processes that access this file even though they may access information related to other accounts. Therefore, all objects created or modified by such processes will become tainted and will unnecessarily be reverted to a previous state. To reduce the possibility of tainting legitimate objects, the Analyzer implements enhancements that relax the application of dependencies. For example, it may choose not to taint a process which reads from the /etc/passwd file.

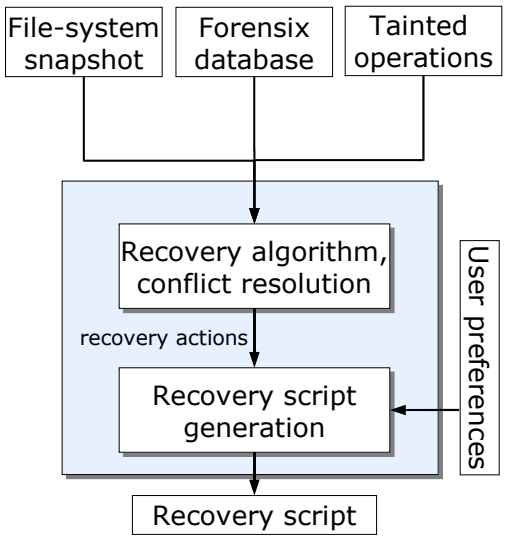

Figure 5.1: The Resolver

The Resolver is the final component of Taser. Its input consists of the set of tainted file-system objects generated by the Analyzer and the entire set of file-system operations available from the audit log in the Forensix database. Figure 5.1 shows the architecture of the Resolver. The recovery algorithms revert tainted file-system objects to a clean state. Our recovery model and the recovery algorithms are described in sections 5.2 and 5.3. The conflict resolution methods handle cases in which a tainted operation cannot be fully reverted because it affects a legitimate object. These methods are described in section 5.4. Finally, the last phase of recovery generates an executable script consisting of a sequence of recovery operations which revert the effects of tainted operations. The next sections describe each part of the Resolver in detail.

5.2 Recovery Model

The recovery model in this thesis assumes a POSIX-compliant Unix file-system consisting of regular files, directories, symbolic links and device nodes, each of which has three types of information associated with it: name, content, and attributes. This model treats file name,

name op	: name id	→ directory name id, name
content op	: object id	→ content
attribute op	: object id	→ attribute

Table 5.2: The recovery model

content and attributes as separate objects during recovery, and assumes that operations on each object are independent. For example, it assumes that name operations occur independently of content or attribute operations. Separating file-system operations helps in optimizing name and attribute recovery as discussed later in section 5.3.

The recovery model distinguishes between a file object and a name object because Unix files can have multiple names. It assumes that an object id uniquely identifies a file object, and a name id uniquely identifies a name object. In Unix file systems, the object id contains the inode number of the file. A name id is associated with exactly one object id and this association is immutable over the lifetime of the system. In contrast, an object id can be associated with multiple name ids because a file object can have multiple names. Additional requirements on these identifiers, such as uniqueness over time, were described previously in Chapter 2. File names in a Unix file system are stored as part of the contents of directories. The Resolver recovers these contents indirectly during the recovery of file names.

The Resolver enforces file attributes such as permissions and ownership at the file object (or inode) level, immaterial of the name by which the file is accessed. For example, modifications to file permissions (e.g., via `chmod`) are assumed to occur directly on the inode rather than via the name of the file.

To formalize the recovery model, we define the three types of file-system operations as the mappings shown in Table 5.2. A name operation (e.g., `rename`) creates, modifies or removes the mapping between a name object and the pair (directory name id, name). A directory name id is a name id associated with a file object of type directory. A content operation creates, modifies or removes the mapping between a file object and its content, and similarly for the

attribute operation. These definitions make the three different types of operations on an object independent of other operations on the same or other objects provided that three *consistency requirements* imposed by the file system are met:

1. The name or object id must exist for a successful operation.

2. The directory name id must exist for a successful name operation.

3. The name mapping must be one-to-one, i.e. two different name objects in the same directory must map to different file names at any given time.

Separating the three different types of operations has two benefits. First, it simplifies resolving conflicts between tainted and legitimate operations, an issue discussed further in Section 5.4. Second, it allows using a more efficient recovery algorithm that is discussed below.

5.3 Recovery Algorithm

The goal of the Resolver is to revert tainted file-system operations but preserve legitimate operations. It takes as input a file-system snapshot, the set of tainted file-system objects generated by the Analyzer, and the audit log created by the Auditor that contains all the file-system operations. The Analyzer marks modification operations to a file-system object that occur after the time the object was tainted as tainted operations. To revert the tainted operations, the Resolver uses a *selective redo* algorithm that only replays legitimate operations in the log that occur on the tainted objects. It assumes that recovery starts with an immutable file system so that the file-system state does not change during recovery.

The Resolver only considers successful legitimate operations that modify the file system; it ignores read-only operations or operations that returned with a failed status. It is possible that that these operations would have yielded different results (e.g., a failed legitimate operation could have succeeded) if the intrusion had not occurred. However, the resolver does not know

the semantics of the processes that issued the legitimate operations, and hence does not attempt to predict process behavior if tainted operations had not occurred. Similarly, the resolver preserves the effects of all legitimate operations even though it is possible that a legitimate operation may have failed if the intrusion had not occurred (e.g., writes to a file made accessible by a tainted operation).

The rest of this section first considers a simple recovery algorithm based on redo logging. Then it presents selective redo, our optimized redo algorithm that is used by the Resolver.

5.3.1 Simple Redo Algorithm

In the simple redo algorithm, recovery starts with a file-system snapshot and sequentially replays the file-system modification operations captured in the audit log. Only the legitimate operations should be replayed since the effects of the tainted operations should be ignored. This simple redo solution is correct because the dependency rules in Table 5.1 ensure that legitimate operations do not depend on tainted operations. Unfortunately, replaying all legitimate operations can be a slow process.

5.3.2 Selective Redo Algorithm

The selective redo algorithm makes two optimizations to improve the performance of the recovery process. First, we observe that the file-system state at recovery time has the correct state for all non-tainted objects. Therefore, the Resolver starts the recovery process with the file system at the recovery time instead of the file system at the snapshot time, and it *selectively* replays legitimate operations *only* on tainted objects. To recover a tainted object, the Resolver obtains an initial version of the object from the file-system snapshot and sequentially replays the object's legitimate modification operations since the snapshot.

A second optimization takes advantage of the recovery model and performs recovery for file name, content and attribute operations *separately*. Separating file-system operations helps

in optimizing name and attribute recovery. At each name or attribute operation, the Auditor captures the complete state of the object as shown in Table 5.2. For example, it captures all the attributes (permission, ownership) of a file after an attribute operation. As a result, a sequence of attribute and name operations can simply be replaced by the last operation during recovery. Therefore, the resolver recovers a tainted attribute or name by replaying the *last legitimate* operation on that attribute or name. In contrast, to recover file contents, the resolver replays all legitimate content operations starting from the snapshot until the first tainted operation. It does so because, for storage efficiency, the Audit log does not store the complete state of the content mapping at each operation. Note that name recovery implicitly recovers directory contents.

The Resolver performs name recovery before content or attribute recovery. This ordering helps meet the consistency requirements discussed in Section 5.2. Intuitively, name recovery sets up a virtual, consistent name space for the recovered file system, and then content and attribute operations are performed on this name space.

Figure 5.2 presents an example to illustrate the selective redo recovery algorithm. This figure shows the snapshot time when the file-system snapshot is taken, the attack time when an attack occurs, and the recovery time when the attack is detected and intrusion recovery is started. All the file-system operations are shown at the top of the figure. This example shows how the file-system operations can be separated into name, content and attribute operations for two files, File 1 and File 2. Operations that occur after the attack time are marked tainted and are shown in boxes. Note that, as mentioned earlier, the dependency rules in Table 5.1 ensure that after the first tainted operation, all operations on a tainted name, content or attribute object are marked tainted.

Recovery starts with the file-system state at the recovery time. Note that File 2 is untainted and no operations need to be redone for this file. In contrast, File 1 has to be recovered. Name 1 can be recovered in a single step by replaying operation 3. Similarly Attributes 1 can be recovered by replaying operation 8. Finally, Content 1 is recovered by replaying operations 5 and 6. In this example, selective redo requires replaying four legitimate operations, whereas the

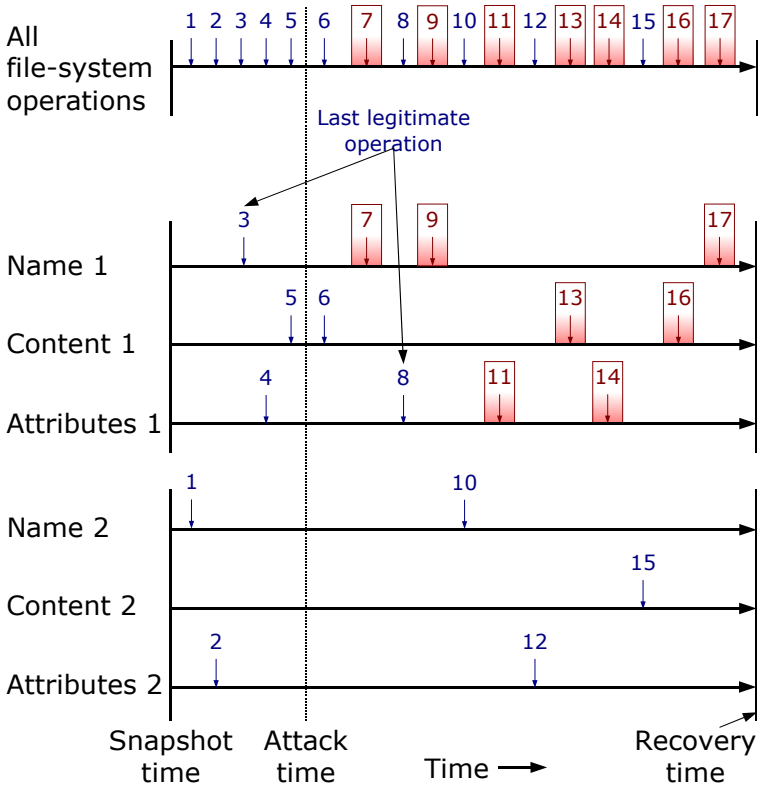

Figure 5.2: Separating content, name and attribute operations

simple redo algorithm requires replaying all ten legitimate operations. In general, selective redo is beneficial if the footprint of the attack is small compared to the total number of legitimate modification operations since the snapshot.

5.4 Conflict Resolution

As mentioned in section 5.1, the Analyzer implements enhancements, that relax the application of dependency rules, to reduce the possibility of tainting legitimate objects. These enhancements can, however, cause *conflicts* when legitimate operations depend on tainted operations. In this case, reverting the tainted operations may result in reverting legitimate file-system operations or the loss of legitimate file-system objects. For example, the Analyzer provides a policy that ignores tainting an object that reads a tainted file or directory name. With this policy, a legitimate file can be created in a tainted directory. If the recovery action for the tainted directory simply removes the directory, then the legitimate file will be lost. We consider such a recovery action as having failed because the goal of the recovery system is to preserve all legitimate operations.

We say that conflicts occur when an operation reads a tainted file-system object, and this read is ignored by the Analyzer. As a result, legitimate operations can occur after tainted operations as shown in Figure 5.3. As the tainting policies of the Analyzer become more optimistic, they ignore more dependencies, and can cause more conflicts. In the example above, reading a tainted directory does not taint the process or the file creation. This conflict would occur with any of the Analyzer's policies that ignores reading file (or directory) names.

Table 5.3 provides a fine classification of conflicts, which allows designing resolution policies suited for each type of conflict. The file-system operations are divided into elementary operations and the types of conflicts are based on these operations. A conflict arises when a legitimate operation, shown along the top row, reads an object that was modified by a tainted operation shown along the left column. Directory-related operations are not shown in the table

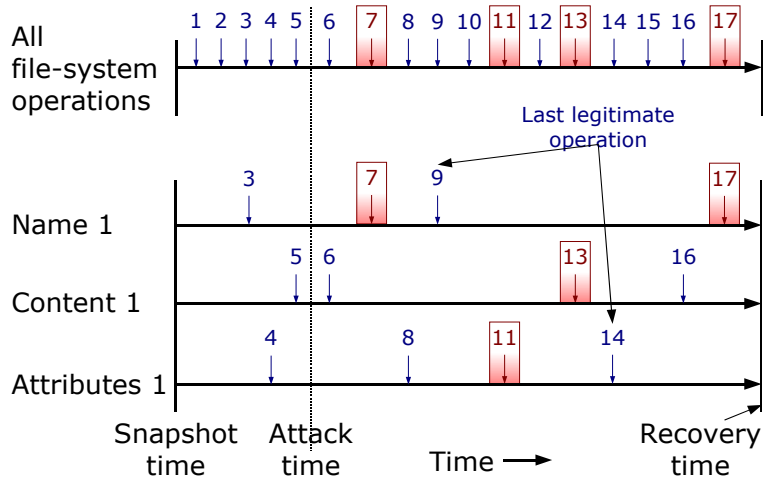

Figure 5.3: Legitimate operations occur after tainted operations

	Legitimate Operations			
Tainted Operations	**Name Create**	**Name, Object Remove**	**Content Update**	**Attribut**
Name Create	name-create conflict	name-remove conflict	name-access conflict	name-acc
Name, Object Remove	name-recreate conflict	not possible	not possible	not p
Content Update	non-conflicting	content-remove conflict	content-access conflict	non-co
Attribute Update	attribute-access conflict	attribute-remove conflict	attribute-access conflict	attribute-ac

Table 5.3: Types of conflicts caused by different legitimate and tainted operations

but are discussed below.

Next, we describe the different types of name, content and attribute conflicts, and the resolution policies implemented by the Resolver. The resolution policies help *isolate* conflicting tainted operations because these operations cannot be completely reverted.

5.4.1 Name Conflicts

Name-create conflict A name-create conflict occurs when a legitimate name creation operation accesses a tainted name. For example, an administrator renames a file created by an attacker. Recall that a tainted name is recovered by simply replaying the last legitimate operation on that name. This operation may conflict with a previous tainted name operation. For example, in Figure 5.3, operation 9 may generate a name that depends on the name produced

by tainted operation 7. This name conflict occurs with any policy of the Analyzer that ignores dependencies caused by reading tainted file names. The Resolver also ignores this conflict because the conflict does not violate any of the consistency requirements described in Section 5.2.

Name-remove conflict A name-remove conflict occurs when a legitimate operation removes a tainted name or a directory containing a tainted name. These conflicts could be ignored because the object was legitimately removed previously. However, it is possible that a user would not have removed this object if the name had not been tainted. For example, the user may have removed a legitimate file that was renamed to an unusual name by a tainted operation. Hence, for these conflicts, the object is recreated with a name that has a `.removed` extension so that it can be inspected manually.

Name-access conflict A name-access conflict occurs when a legitimate operation updates the content or attributes of a file with a tainted name, or modifies a file under a tainted directory. In this case, the file or the directory has seen no legitimate name operations. This object should be removed, but the relevant legitimate operations should be recovered. For example, an administrator may have created a legitimate file under a tainted directory. Simply removing the tainted name would violate one of the first two consistency requirements described in Section 5.2. To resolve this conflict, the tainted name of the object is isolated, instead of being reverted, and recovered with a `.nonexistent` extension. This extension indicates that the name was not created legitimately. At the end of recovery, a list of these suspect objects is provided so that the user can inspect these objects and take appropriate actions.

Name-recreate conflict A name-recreate conflict occurs when a legitimate operation recreates a name that was removed by a tainted operation. For example, the administrator may recreate a legitimate file that was removed by an attacker. Simply recreating the removed tainted object leads to two different but legitimate objects with the same name, which violates the third consistency requirement. If the recovery action recreates the same object such as via

multiple names of a file, then the conflict is ignored. Otherwise, the resolver recreates the previous objects with the same name but with a version number extension.

5.4.2 Content Conflicts

Content-access conflict A content-access conflict occurs when a legitimate operation updates the tainted contents of an object. Recall that the resolver replays the legitimate content operations starting from the snapshot until the first tainted operation. For example, operations 5 and 6 would be replayed to recover Content 1 in Figure 5.3. Any legitimate operation after the first tainted operation causes a content-access conflict because the Resolver assumes that content operations always read contents before modifying them. Content-access conflicts need to be fixed manually since file contents are typically unstructured. An alternative is to use application-specific conflicts resolvers [26, 20, 37].

Content-remove conflict This conflict occurs when a legitimate operation removes an object whose content is tainted. Similar to the reasons for storing objects that are involved in a name-remove conflict, objects involved in a content-remove conflict are also recreated with a name that has a `.removed` extension so that the contents of the object can be inspected manually.

5.4.3 Attribute Conflicts

Attribute-access conflict An attribute-access conflict occurs when a legitimate operation (other than remove) accesses the tainted attributes (permission or ownership) of an object. Recall that a tainted attribute is recovered by simply replaying the last legitimate operation on that attribute. For example, Figure 5.3 shows the last legitimate and the last tainted operations (operations 14 and 11) on Attributes 1. In this case, since the last operation is legitimate, nothing needs to be done for recovery, otherwise, operation 14 would be replayed if there were tainted operations after it. Similar to name-create conflicts, the resolver ignores this conflict because it does not violate the consistency requirements.

Attribute-remove conflict This conflict occurs when a legitimate operation removes an object whose attributes are tainted. This conflict is resolved in a similar way to name-remove conflicts by recreating the object with a name and that has a `.removed` extension and with legitimate attributes.

5.4.4 Global Conflict Resolution

Recall from Section 5.2 that the Resolver performs name, content and attribute recovery separately. Typically, conflict resolution is performed as part of the corresponding recovery operation. For example, name conflicts are resolved as part of name recovery, etc. However, certain conflicts must be resolved globally after all recovery actions have been generated. For example, a name-access conflict occurs when a legitimate operation updates the content or attributes of a file with a tainted name. This name conflict can be detected and resolved only after the name, content and attribute recovery algorithms have been executed. In particular, the tainted name cannot be removed during name recovery if the object has legitimate content or attribute modifications.

5.5 The Resolver Implementation

This section describes the implementation of the the Resolver component of the Taser intrusion recovery system. The recovery algorithms in the Resolver selectively recover a file-system after an intrusion by reverting the tainted file-system operations. The Resolver also contains conflict resolution methods which ensure that all legitimate data is preserved after recovery. Below, we describe the structure of the Resolver and its implementation.

5.5.1 Resolver Structure

Figure 5.4 shows the structure of the Resolver. The input to the Resolver is the Forensix auditing database that stores a file-system snapshot and the file-system operations since the

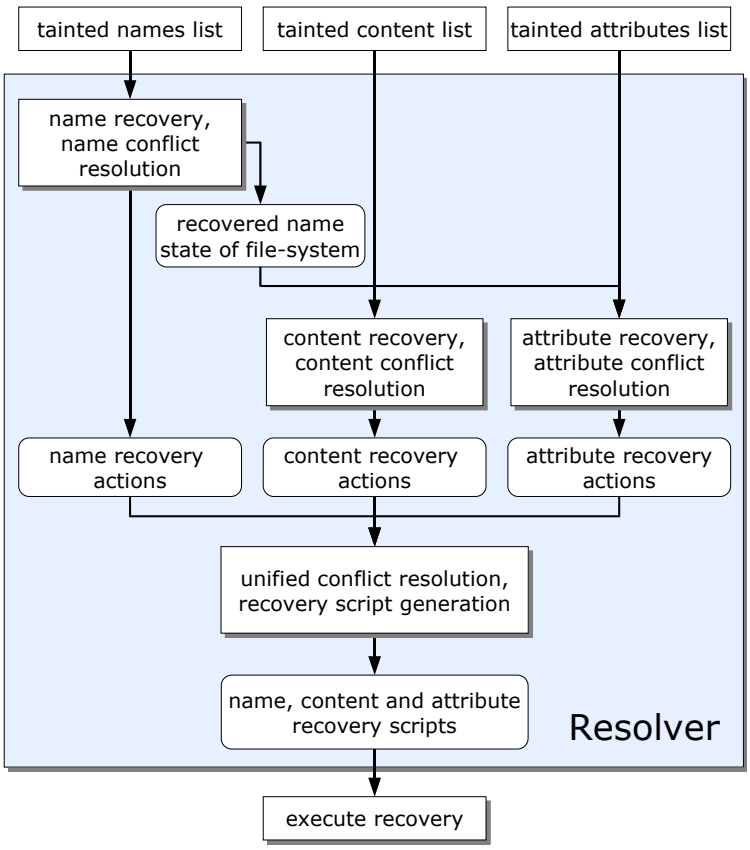

Figure 5.4: Structure of the Resolver

snapshot, and a table of tainted objects and operations that are provided by the Analyzer. The Analyzer provides a separate list of tainted name, content and attribute objects as shown in Figure 5.4. Table 5.4 shows the different types of file-system operations considered during recovery. At the end of recovery, the same types of operations are issued as recovery actions to revert tainted operations.

The name, content and attribute recovery algorithms are performed together with the corresponding conflict resolution methods. Note that name recovery is performed before content and attribute recovery because it sets up the name space before the other recovery actions are applied to the file-system. The output of the recovery algorithms are recovery actions that take the immutable file system at recovery time to a recovered system that has reverted or isolated the effects of tainted operations. The recovery actions are fed to the global resolution phase

	Create	Update	Remove
File names	create, mknod, link, symlink	rename	unlink
Directory names	mkdir	rename	rmdir
File contents	create, mknod, symlink	write, truncate	n/a
File/directory attributes	create, mknod, mkdir, symlink	chmod, chown	n/a

Table 5.4: File-system operations

that performs the unified conflict resolution operations.

A final recovery script generation phase orders the recovery actions so that they can be executed consistently. For example, suppose name A at recovery time must be renamed to name B. However, name B exists at recovery time and must be renamed to name B.old. Then the script generation phase orders the second action before the first one. This phase takes user preferences into account such as whether old object versions should be kept or certain files can be ignored for recovery (e.g., editor backup files). The Resolver runs *entirely* on the backend system and then the recovery scripts are executed on the target.

The Resolver uses some of the interval tables described in Section 3.2. These tables store an archive of the mapping information described in Table 5.2, and they allow creating snapshots of the file system, including the state of the file system just before the attack time or at the recovery time on the backend machine.

Next we describe each component of the Resolver in detail. For implementation purposes, we define two new terms. The last legitimate and tainted operation times T_l and T_t are the times at which the last legitimate and tainted operation made a modification to the object. There are separate T_l and T_t times for name, content and attribute operations. The time T_l (or T_t) is marked as `null` when no legitimate (or tainted) operation has ever been performed on the object. Note that the recovery algorithm only considers objects that have been modified by a tainted operation, and hence the time T_t for at least one of the name, content or attribute operations is not `null`.

5.5.2 Name Recovery

The name recovery code is shown in Figures 5.5 and 5.6. At a high level, the first part of the algorithm deals with conflicts and appends tainted names to a recovery list that consists of all names that need to be recovered. The recovery action itself is performed at the end in the `finalize_name_recovery` function. Names are appended to the recovery list using the function `get_name(name id,` T_l`)`, where name id is the unique identifier described in the beginning of this chapter. The `get_name` function returns the tuple (name id, T_l, name, parent-dir id) where the name and the parent-dir id are the name and the directory containing name id immediately after time T_l (or before T_l, when T_l refers to a remove activity). This function is implemented by querying the current recovery list and the file system at the recovery time T_l, which together represent the "recovered name state" at recovery time. Note that searching a name id in the file system is performed by issuing a query to the inode interval table in the auditing database. Function `mark_conflict(name id, tag)` simply appends a tag to the name associated with the given name id to show the type or reason of conflict occurred during recovery of this name. At the end of recovery, these tagged names can be looked up and analyzed manually.

The input to the name recovery is the `tainted_name_list` that contains all the tainted names whose time T_t is not `null`. Each name in this list, identified by its name id, has the last legitimate and tainted activity times T_l and T_t associated with it. When T_l is `null`, the name has not been created, updated or removed by a legitimate activity. This name is marked and is removed at the end of recovery if it does not generate a `name-access` conflict. A `name-access` conflict occurs when the object with this name has legitimate attribute or content updates, where for directories, content updates occur when it has objects underneath that have legitimate name, attribute or content updates.

When $T_l > T_t$, no recovery action is required if the name was last created or updated by a legitimate activity since it is assumed to be legitimate now. If the name was legitimately removed (it is not in the file system) then this tainted name is marked with a `name-remove`

```
name_recovery(tainted_name_list):
  recovery_list = empty
  foreach name id in tainted_name_list:
    if T_l is null:
      mark_conflict(name id, name-access)
    if T_l > T_t:
      if name id removed at T_l:
        mark_conflict(name id, name-remove)
        append(recovery_list, get_name(name id, T_l))
    else:
        append(recovery_list, get_name(name id, T_l))

  mark_remove_parent_conflict(recovery_list)
  mark_name_recreate_conflict(recovery_list)

  // all conflicts have been handled
  foreach not done element in recovery_list:
    perform_name_recovery(recovery_list, element)
```

Figure 5.5: Name recovery code (Part 1)

```
mark_remove_parent_conflict(list):
  foreach name id in list:
    if parent-name id not in recovered state:
      mark_conflict(parent-name id, name-remove)
      append(list, get_name(parent-name id, T_l))

mark_name_recreate_conflict(list):
  foreach name id in list:
    if name id has duplicate in recovered state:
      mark_conflict(name id, name-recreate)

perform_name_recovery(list, element):
  if parent-dir id in list:
    finalize_name_recovery(list, parent-dir)
  generate_name_recovery_action(element)
  mark_done(list, element)
```

Figure 5.6: Name recovery code (Part 2)

conflict and appended to a recovery list. In this case, recovery will involve recreating the name just before it was removed. If the remove was for the last name of an object, the object is also recovered. Finally, when $T_l < T_t$, the name is appended to the recovery list so that it can be recovered.[1]

The function `mark_remove_parent_conflict` checks that all parents of a name that needs to be recovered exist in the recovered state, i.e., either in the file system or in the recovery list. If not, the parent was removed legitimately but with a `name-remove` conflict. Therefore it is appended to the recovery list with a name-remove tag so that it can be recreated. This function ensures that all parents that are needed for correct recovery will eventually be in the recovered state. The function `mark_name_recreate_conflict` checks for names in the recovery list that have duplicates in the recovered state. A duplicate occurs when, in the recovery-list and file-system at recovery time, there exist two or more name ids with the same pair (name, parent-name id). For example, the file /A/B exists and there is a recovery action to create another /A/B. In this case, except the name id with the latest T_l, all the others are marked with the `name_recreate` conflict. At the end of the name-recovery, these names are either removed or recreated with a `version` extension depending on the policy used for name-recreate conflicts.

At this point, all name conflicts have been handled, and the function `perform_name_recovery` performs recovery recursively from the top to the bottom of the name hierarchy. The function `generate_name_recovery_action` generates the actual name recovery action, which can consist of either creating the first name of an object together with the object, or renaming of an object, or creating an additional name for an object. When an object is created, its contents and attributes are recreated later as part of content and attribute recovery. At the end of recovery, objects marked with the name-access tag (nonexistent) are removed if they do not have `name-access` conflicts. Also, versioned objects are handled depending on the

[1]Note that in this case the last legitimate activity can only be a name creation or update. It cannot be name removal or else $T_l < T_t$ would not be true and so the name should be recovered.

```
for each object id in tainted_content_list:
  if object id not in recovered state:
    mark_conflict(object id, update-remove)
    append(recovery_list, get_name(object id, T_l))
  generate_content_recovery_actions(object_id)
```

Figure 5.7: Content recovery code

name-recreate policy.

One optimization to the entire recovery code is to implement an efficient `get_name` function which is used frequently in all the recovery algorithms to lookup path names of different name ids at either recovery time (current path name) or at recovered name state (target path name). As described in section 4.3, resolving a name id to path name is an iterative process which builds the path name component by component. To optimize this function, we can implement two caches which store lookups of directory components of path names at recovery time and at the recovered name state. For example, after resolving a name id to path name /A/B/C at recovery time, any other name id in directories /A and /A/B at the same time can be resolved to its path name directly using the corresponding cache.

5.5.3 Content Recovery

The input to the content recovery algorithm shown in Figure 5.7 is a set of content-tainted file objects along with all content-modifying legitimate or tainted operations. First, the code checks whether a tainted object exists in the recovered state. If not, the object is marked with an `update-remove` conflict since this object has been removed by a legitimate activity but has tainted content updates. Such an object is recreated, but since it may never have had a legitimate name, we recover such objects in a separate `orphanage` area and add it to the recovery list with a generated name so that later, if needed, attribute recovery will find this object in the recovered state.

Content recovery actions consist of legitimate content-modifying operations until T_l. Unlike name and attribute recovery, which are assumed to be atomic and non-conflicting, content

modification can involve partial reads and writes. Recovery only redoes those operations whose target position in the file is not dependent on any previous tainting operation and hence all legitimate appends are redone. Further, a legitimate truncate followed by legitimate writes until the first content tainting activity are also redone. All other legitimate modifications are reported as content conflicts that must be resolved manually or via application-specific resolvers.

5.5.4 Attribute Recovery

The attribute recovery code is very similar to content recovery and not shown. The only differences are that attribute recovery is performed only when the last legitimate activity occurs before the last tainted activity ($T_l < T_t$), and it is a one-shot operation that sets the attribute to the attribute value at T_l.

5.5.5 Global Conflict Resolver

The global conflict resolver performs some final recovery actions by identifying the actual `name-access` conflicts. Having all the recovery actions from name, content and attribute recovery phases, it finds the tainted names that have a `name-access` conflict tag and have no legitimate attribute or contents for their objects at the end of recovery. [2] These tainted names do not cause a `name-access` conflict. Therefore, their conflict tag is removed and any of them which still exists on the system (at the recovery time), will be removed by issuing a remove action for it.

[2]For a directory object, this means that the directory's attributes are still tainted and the directory is empty in the recovered name state. However, a directory may become empty as part of the global conflict resolution (e.g. its only file is deleted). To solve this issue, the global resolution is performed from the bottom to the top of the directory hierarchy.

5.5.6 Recovery Script Generator

The input to the recovery script generator consists of a sequence of commands such as (recovery_action, current_path, (optional) target_path, options) produced by the name, content and attribute recovery phases. The target_path, obtained from the recovered name state of file-system, is the final path of an object after the entire recovery has taken place. At this point, a user has the choice to observe the list of recovery actions and remove any undesired command. Finally, all remaining recovery actions are parsed and reordered such that they can be executed correctly on the target system and without violating any file-system consistency requirements (described in section 5.2). For example, a file should be created after and not before the creation of its parent directory. Or, the renaming of file X to Y and of file Y to X should be done using a temporary name such as Z.

An important issue with the final executable recovery script is that it should not depend on the target system. For example, the recovery script is a statically linked executable that does not use any library on the target system since these libraries may have been compromised themselves [38].

Chapter 6

Evaluation

This chapter evaluates the functionality and performance of the intrusion analysis tools and the recovery algorithms presented in this thesis. We present our analysis results for a target system that was attacked multiple times during the course of a week. The results demonstrate how intrusions can be interactively and easily analyzed using the set of tools implemented in this thesis. Then, we evaluate the performance overhead of the backend system. Finally, to evaluate our recovery algorithms, we run the Taser recovery system [13] to analyze real attacks and evaluate the performance cost and correctness of the recovery performed by the Resolver.

6.1 General Setup

The experimental setup consists of a target machine where Forensix [12] audits and logs Linux kernel events. This data is streamed to a backend system where it is periodically loaded to a MySQL database. Then the interval tables are updated to reconstruct system state and, at this point, we can perform intrusion analysis and recovery. The target and the backend machines both run AMD Athlon MP 2600+ machines with 512 MB RAM. The target runs stock RedHat 7.2 together with the Forensix auditing module. It contains four vulnerable services or executables: the samba and the wu-ftpd daemon that allow remote root exploits, and the sendmail and the pwck-setuid programs that allow local root escalation exploits. The experiments use

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 only allows an authenticated connection from the target machine. The backend machine runs Redhat Fedora Core 3 and uses the MySQL version 4.1.10 database for storing the audit data.

6.2 Intrusion Analysis Evaluation

The target was run with the vulnerable services for approximately a week from May 11th until May 18rd, 2005. During this time, an external attacker (unknown to us) successfully gained access to the target by using the Wu-ftpd remote root exploit around 5pm on May 12th. The following subsection presents analysis of the ftpd attack. During the week, several other attacks were run on the system and later analyzed in detail. The analysis methods for these attacks were similar to the ftpd attack and not presented here.

6.2.1 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 shell-code. 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 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 6.1, 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 since system files such as /usr/bin/killall have been modified.

Next we queried for the process which created the new /usr/bin/killall and, using the query to find the parent of a process, traversed up the process hierarchy to find ftp daemon. Querying the `process` and `process_owner` interval tables revealed the spawned process is a non-

```
/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 6.1: File-access tracker output for ftpd attack.

```
/usr/sbin/adduser pol
passwd pol
wget XXXX.XXXXXXXXXX.com/kanaris/adrian/rk.jpg
tar xzvf rk.jpg
./setup
rm -rf rk.jpg
```

Figure 6.2: Attack activities before getting the interactive root shell

interactive root-shell which we replayed it using our shell-IO tracker tool. The shell commands are shown in figure 6.2 in which we see the creation of a user account pol and the downloading of the rk.jpg file which contained the rootkit. The file was untarred and removed later.

Using the file-contents constructor, we recreated the removed rk.jpg file that installs a backdoor. To find out more about the backdoor, we issued a query on the `connection` and the `process` interval tables. This query revealed ports that had been opened between 17:00 and 18:00, and here, we found a process called sendmail that was listening on port 212 from 17:12 and was probably used to run an interactive shell. Therefore, we then queried the `inode` interval table for any instance of /dev/pts/x creations between 17:00 and 19:00. This query returned one row that showed an interactive shell was used from 17:12 until 18:40. A query to the `process_owner` interval table showed that this attacker's shell was also run as root. Next we used our IO tracker tool to replay the shell. Key output from the shell session is shown in Figure 6.3.

The psyBNC.tgz file, which is removed by the attacker and which we later recreated, has an executable file disguised as crond. The attacker runs the SucKIT rootkit that does not need

```
[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
```

Figure 6.3: IO tracker output for the ftpd attack.

a kernel with support for loadable kernel modules [29] but is loaded through /dev/kmem into the kernel. With the rootkit, the attacker tried to hide the fake crond process but since we use LIDS [41] on the target system to disable writes to /dev/kmem, the attacker was not successful.

6.2.2 Analysis Results

The previous section described the queries we ran to analyze the ftpd attack. The total time taken to run each of the queries is shown in Table 6.1. This table shows that all these queries that use the interval tables run quickly and can be used by an interactive user.

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
Traverse the process hierarchy up to the source of attack	2 s
Finding uid of the shell process	< 1 s
Replaying attacker's shell	1 s
Recreation of the removed attack files	3 s
Finding the interactive shells	< 1 s
Finding the listening port set by the attack code	< 1 s

Table 6.1: Time taken for each intrusion analysis query.

Without the interval tables, historical analysis queries on the Forensix event data 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 6.1 without using 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.

6.2.3 Performance Measurements

Forensix logs data at the frontend machine and loads this data at the backend machine. Logging imposes overhead at the frontend while loading, storing and analyzing the data imposes overhead at the backend. Analysis and recovery are performed entirely on the backend machine and hence we present the performance overheads on the backend machine. Note that the frontend performance is mostly decoupled from the backend performance. For example, the analysis queries in the previous section are run at the backend system and have virtually no affect on the performance of the frontend system. For the frontend performance measurements, the reader is referred to the original Forensix work [12].

The target system was run with the vulnerable services for approximately a week. During this time, we ran a popular web-based photo album application called Gallery on the target system. To load the system, we simulated users' interactions with Gallery with a client-side Galhogger program. Galhogger browses the photo albums 75% of the time and modifies the albums rest of the time. Browsing reads image files and modifies statistics files. Album modifications include 1) adding and deleting photos which creates and removes image files and 2) adding and deleting albums which creates and removes directories and some other statistics files. The program was run non-stop for the entire week, simulating 5 heavy users that performed an operation every 4 seconds on average.

The cost imposed at the backend, averaged per day, is shown in Table 6.2. The system load and hence the daily numbers do not vary much across different days. The figure shows the

number of kernel events generated on the target machine. The most common events consist of `open` (3.3 M), `close` (2.5 M), `read` (6.1 M), `write` (540 K), `create` (240 K), `unlink` (160 K), `connection` (36 K), `exec` (24 K) and `fork` (16 K) events. These constitute 83% of all events.

Number of Events	15.6 Million
Loading time	37.8 min
Interval table generation time	26.4 min
Size of events in flat file	1.7 GB
Size of database w/o interval tables	2.2 GB
Size of interval tables	64 MB

Table 6.2: Average daily backend statistics

The generation of the interval tables imposes a 70% overhead (26.4/37.8) at the backend. Another way to interpret the results, based on the loading time of 37.8 minutes, is that the Forensix backend, without the interval tables, would be able to sustain a 38 time larger load (24*60/37.8) than our experiment load. With the interval tables, the load could be 22 (24*60/(37.8+26.4)) times larger than our experiment load. While the generation time for the interval tables is not small, it is comparable to the batch loading of files in a database, and becomes relevant only when the backend load gets very high.

Note that the size of the interval tables is small compared to the original database (it adds about 2.9% overhead (0.064/2.2)). Queries using the interval tables access less data, which partially explains why they can be run faster. Table 6.2 shows results for an entire day. However, note that the database is loaded every two hours in our system and can be loaded more frequently for intrusion analysis with shorter delays.

While the storage requirements of the Forensix system are large, the large amount of network capacity and massive and inexpensive storage space available in local networks today (a terabyte costs between $500-$1000) make the Forensix approach feasible and essential for reliably analyzing and recreating intrusion activity.

6.3 Intrusion Recovery Evaluation

In this section, we evaluate the functionality and performance of the recovery system. We use various exploits to attack the target system. Each of the exploits gain access and affect the file-system. Our recovery system reverts effects of those attacks. During each attack and before recovery, legitimate activities are issued to simulate load on the system. This is done using the Galhogger program described in the previous section. For each attack, we used the Analyzer component of Taser [13] to provide a set of tainted objects and operations as input to the Resolver. Then, the Resolver generated a recovery script which contains a list of recovery actions that revert the tainted operations and resolve conflicts caused by legitimate operations that depend on some tainted operations. For recovery actions, we count the number of distinct name and attribute recovery operations and the number of distinct file objects that require content recovery. Below, we describe three scenarios and the recovery actions generated for each of them. Table 6.3 shows the scenarios, the number of recovery actions, the number of legitimate modification actions during and after the incident, and the recovery time. The number of legitimate actions is the total number of file-system modification operations that have occurred on the system since the attack time.

Scenario	Recovery Actions	Legitimate Actions	Recovery Time
Content destruction	739	5338	9 s
Compromised database	1617	2557	21 s
Software installation	350	5006	5 s

Table 6.3: Recovery measurements for different scenarios

A. Content destruction

Scenario: 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 project.c file in `backup/project.c.bak`. Another developer on the system launches the sendmail local escalation exploit to get the root shell. This attacker deletes the `project.c` and `p2.h` files. The victim notices that the `project.c` file is missing and copies the backup file to the `src` directory.

Then he moves the `p1.h` file to the `src` directory and deletes the `hfiles` directory.

Recovery actions: Remove numerous files generated by the sendmail attack, restore the deleted `p2.h` and `project.c` files. The `hfiles` directory is recovered with a `.removed` extension since it was legitimately removed by the victim (a name-remove conflict). The original `project.c` file is recovered with a version extension since a legitimate file with the same name already exists (a name-recreate conflict).

B. Compromised database

Scenario: Authenticated MySQL clients update a MySQL database running on a remote server. An attacker launches a remote attack on the Samba daemon running on the system, gets a root shell and creates an SSH backdoor by writing his public key to root's `authorized_keys2` file. He also downloads and installs a rootkit. Later, other remote legitimate clients insert transactions into the database. After six hours, the attacker uses the ssh backdoor to log back into the machine. He issues a local MySQL query to remove some transactions from the database. After that, more legitimate clients update the database.

Recovery actions: Remove the attacker's ssh backdoor by removing his public key from the `authorized_keys2` file. In addition, remove rootkit files and recover two files associated with a MySQL table in the compromised database. The database files are restored to the state right before the attacker's second login when he modified the database. The rest of the legitimate writes were marked with a content conflict but we were unable to recover them. For transactional databases, the database would need to be involved in the recovery process [21, 23].

C. Software installation

Scenario: This scenario, unlike the previous ones, presents and analyzes system administration error. Using a root account, the system administrator installs RealPlayer 8 in the wrong directory which causes it to create many files and directories in this directory. In addition, it creates or updates various Netscape, KDE and Gnome configuration files or directories in

`/root` including a `.netscape/plugins` directory. Later, the root user browses the web with the netscape browser and downloads and saves a PDF reader plugin for Netscape in this directory.

Recovery actions: Remove all the RealPlayer files and directories and restore the configuration files. However, the plugins directory is recovered with a `.nonexistent` extension because this directory, created by the tainted process, contains the legitimate plugin file (name-access conflict). The user needs to retrieve this file separately.

In summary, in all the above scenarios, the recovery system generated recovery scripts that correctly reverted the effects of the tainted operations, and the scripts handled conflicts so that legitimate operations are preserved as much as possible. Also, recovery takes a short time in the order of seconds.

Chapter 7

Related Work

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

7.1 Intrusion Analysis

Current intrusion analysis tools provide piecemeal or "lossy" information and hence lack the ability to accurately reconstruct what happened in the system. For example, application and system log files only track events based on what the applications and system administrators think is necessary to log. On the other hand, process accounting mechanisms only provide information as to how commands are executed, and can fail to track what programs are doing internally or the sequence of activities that led to an attack. Network traffic traces alone are also problematic in that sessions can be encrypted and it is extremely difficult to correlate network forensic information directly to higher-level application behavior that elucidates the actual damage done to the target system. File system activity logs can only detect modifications to files and thus are unable to address attacks in which running processes are compromised directly [6]. A secondary problem with many existing intrusion analysis and detection tools is that they can affect evidence when conducting analysis. For example, accessing a file on the system changes its access time.

In this thesis, we use the Forensix [12] audit system that uses kernel-based logging. An alternative is to use Virtual Machine(VM)-based auditing that can provide additional resistance to attacks on the logging mechanism. ReVirt [7] places a system within a virtual machine and logs the VM-to-host instruction stream and allows replaying and analyzing the system's execution before, during and after an intrusion. ReVirt replays activity in linear time order which can slow down the analysis. However, this approach 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 Virtual Machine Monitor(VMM)-based kernel introspection mechanism [11]. Garfinkel [10] demonstrated several races in using system call interposition. To resolve these races, Forensix uses the Linux security modules facility [40] to correctly order events in time and unambiguously identify system objects (e.g., files).

System call traces have been used in the past to identify normal system behavior and to automatically detect suspicious behavior or intrusions [16, 30, 33, 8]. However, these approaches examine system-call patterns over a short window of 5-100 calls and hence do not completely capture system activity for intrusion analysis. In contrast, Forensix captures system calls, their timing, parameters, return values and the user making the call for long periods of time for analysis.

Sandboxing techniques are complementary to the intrusion analysis approach described in this thesis. They interposition code that allows blocking program actions that may compromise security. Janus [14] interpositions system calls using the `proc` file system. Systrace [25] 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. It should be possible to implement sandbox policies using our analysis tools.

Tripwire [18] monitors the cryptographic hash and size of key system files and directories and reports file accesses and modifications. However, it does not provide information about the processes or users that modified the files. Venema and Farmer have developed the Coroner's

Toolkit (TCT) [9] that can be used for postmortem analysis of a UNIX system after a break-in, similar to our system. TCT provides tools to collect forensic information from different sources (e.g. memory, file-system, system utilities and logfiles). These tools can reconstruct sequences of past events. With Forensix, this analysis can be performed using simple queries to the database. The TCT `mactools` program that displays access times and patterns of open, currently accessible or deleted files motivated our file-access tracker tool. To recover deleted files, TCT implements the `unrm` and `lazarus` tools that try to recreate files from unallocated disk blocks. The tools described in this thesis perform all these tasks and do so accurately since we store the system state including all writes over time. To do so, one must necessarily audit more information, but as explained in Section 6.2.3, storage space is abundantly available today. Unlike TCT, we use a separate secure backend which allows storing the audit data more securely and avoids disturbing the target machine during analysis. Interestingly, the Forensix auditing system automatically captures the output of other analysis tools running on the target machine so we can benefit from these tools, e.g., generate the output files of sysstat, a system monitoring utility, in the last three days.

The Sleuth Kit [4] is a derivative of Coroner's Toolkit and provides file system information, file names and contents from file inode information and lists recently deleted files in a directory. Simple analysis queries in Forensix can provide this information without requiring any knowledge of the file system structure. Sebek [17] captures the `write` system calls so that it can replay an attacker's keystrokes similar to the shell-IO tracker tool. Since only the `write` calls are captured but not the inode number to which the write occurred, the replay can produce significant irrelevant information as well as miss information. Sebek is mainly intended for honeypots so it takes detailed measures to hide its presence, which unfortunately can lead to it being used *against* unsuspecting users. Our goal is to implement a secure but visible auditing system that acts as an intruder repellent.

Complementary to the host-based intrusion analysis tools in this thesis are network-based analysis tools such as SNORT [27] that capture and log network packets and, in addition, 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.

7.2 Intrusion Recovery

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 [28] uses a clever purging method that keeps "landmark" data versions and purges generated and temporary files aggressively, while CVFS [35, 34] encodes meta-data versions efficiently. The recovery system described in this thesis uses an unoptimized data storage mechanism and would benefit from some of these techniques, although purging data versions would limit some of the benefits of the recovery approach. 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 [42] has goals closest to this work. It supports fine-grained logging to allow roll-back of any file update operation, and has a basic contamination analysis mechanism to determine the extent of system damage after an incident. However, during recovery, this file system does not seem to consider conflicting operations, which if not handled correctly, may lead to loss of useful data. Application-specific conflict resolution policies has been extensively studied in the context of replicated file systems [26, 20] and databases [37]. While this work has not experimented with these policies, they would directly apply to our conflict resolution techniques.

Fastrek [23] recovers 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 operation is rolled back if it may have depended on the data produced by a tainted

operation. While conservative tainting of file-system operations effectively achieve the same result in our approach, the conflict resolution policies allow using optimistic taintings that reduce this problem. Liu et al. [21] describe algorithms that rewrite transaction history in a database by moving the attacking transaction and all affected transactions after non-affected transactions.

Brown [3] describes an application-neutral framework which allows operators to recover from their own mistakes. Their proof-of-concept implementation provides a recovery service that deals with operator errors in a mail server. While it is possible to extend the service to other applications, it is unclear how much effort is involved since it has to be tailored for each specific application. In contrast, our system is geared towards recovery at the file-system level which does not necessarily have the clearly defined semantics of a mail server and hence our recovery techniques are more generic.

Sun [36] 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. In addition, rollback caused by consistency criteria being violated, becomes more likely for long running SEEs.

Our approach of recovering only the compromised parts of a file-system while preserving legitimate data has similarities with system software upgrade where the upgrade can be selectively rolled back without affecting the rest of the system [1, 15]. Also, non-linear and selective undo recovery have been extensively studied in collaborative application environments. Prakash and Knister [24] propose a general framework for undoing an individual's actions in collaborative systems based on defining inverse, conflict and transpose functions. Berlage [2] describes a technique in graphical user interfaces (e.g., editors, paint programs) where selec-

tive undo only looks at the command to undo and the current state, and does not depend on the history in between.

Chapter 8

Conclusions and Future Work

Currently, after a computer system is compromised, the user must manually sift through large amounts of uncorrelated system and application-specific log files to understand what happened on their system. This process is tedious and highly error prone, does not provide a complete picture of the attack, and makes it hard to retrieve uncorrupted data. Further, it does not necessarily provide hints on how the system can be hardened. To deal with this problem, this thesis uses a complete system-level audit trail for intrusion analysis. The challenge with such approach is to provide tools that allow analyzing the large amount of data that is generated. This thesis shows that interval tables, which contain lifetimes of system objects and their attributes, allow interactive analysis of large data sets. With these interval tables, we have implemented a number of powerful, efficient and easy-to-write intrusion analysis tools and used them to analyze real attacks in detail.

This thesis also focuses on intrusion recovery. The current approach of using snapshot-based file-systems to recover from intrusions or from human errors is well understood and easy to use but it works well only when intrusions (or errors) can be immediately detected. Otherwise, a snapshot before an attack loses legitimate user modifications that occur after the attack and that data needs to be recovered manually. We describe a framework that recovers persistent data after an intrusion or local damage occurs. This framework performs recov-

ery efficiently by using a redo approach that reverts the effects of intrusions. The framework also provides conflict resolution methods to handle conflicts caused by dependencies between legitimate and tainted operations. The resolution methods ensure that all legitimate data is preserved by isolating the tainted objects involved in a conflict. This framework is implemented as part of the Resolver component of Taser [13]. Our evaluation of the Resolver shows that recovery actions can be performed quickly for a wide range of intrusions as well as erroneous user action scenarios.

8.1 Future Work

There are two main areas of research related to intrusion analysis and recovery that we plan to pursue in the future. Below, we discuss each area separately.

8.1.1 Enhanced Intrusion Analysis

The tools implemented in this thesis enable analysis of an intrusion to find out what happened on the compromised system. These tools could be enhanced with graphical tools. For example, the Backtracker [19] is a graphical tool that uses a timing-based approach to generate dependencies between processes, files and filenames and uses the dependency graph to view intrusions. Similarly, an attack graph [31] represents the steps of the attack and the system or network vulnerabilities used by the attack. Attack graphs help to characterize the attack and perform vulnerability analysis on systems. Using our analysis infrastructure we hope to define and implement similar graphical tools which provide specific views of system events.

We believe that our analysis tools can be used to improve current intrusion detection techniques. Currently, different intrusion detection systems (IDS) monitor different parts of a system (e.g. file-system, network, kernel, etc.). The problem with these IDSs is that they are not integrated and each IDS can generate a large number of false alarms. It should be possible to use multiple IDS sources so as not to miss any attack and then correlate the alarms using our

intrusion analysis tools.

Cross-machine intrusion analysis is another area which needs to explored in future. The main problem which we currently see in this area is that in general, operating systems provide little or no support for tracing user activity across machines. For example, the username, shell process id or connection identifier on the current machine is not reliably transfered to the second machine upon connection. Our intrusion analysis tools could be enhanced to perform cross-machine analysis with this type of information.

8.1.2 Towards Automating Recovery

We believe the recovery framework defined in this thesis provides the basis for developing automated intrusion recovery solutions. In the future, we plan to explore whether the recovery system could be integrated with journaling and versioning file systems to improve scalability and to reduce disk space requirements. Another potential is to add recovery support for network file systems such as NFS. The reason this is not currently done is that the recovery system audits at the client end which causes consistency issues for concurrent accesses. Auditing at the server end would avoid this problem. Finally, we plan to implement a common recovery infrastructure that can be used by complex application programs such as databases so that they can be recovered after an intrusion. Such applications will benefit from an application-specific recovery policy, which ideally, should be provided by the application itself to the recovery system.

Bibliography

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

[2] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

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

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

[5] CERT Coordination Center. Cert/cc statistics 1988-2004. `http://www.cert.org/stats/cert_stats.html`.

[6] Crispin Cowan. Immunix: Adaptive system survivability. `http://www.immunix.org`, 1998.

[7] 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.

[8] 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.

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

[10] Tal 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.

[11] 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.

[12] 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).

[13] 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.

[14] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, 1996.

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

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

[17] The Honeynet Project. Know your enemy: Sebek. `http://www.honeynet.org/papers/sebek.pdf`.

[18] 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.

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

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

[21] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[22] N. Nguyen, P. Reiher, and G.H. Kuenning. Detecting insider threats by monitoring system call activity. In *IEEE Information Assurance Workshop*, New York, June 2003.

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

[24] Atul Prakash and Michael J. Knister. Undoing actions in collaborative work. In *Proceedings of the ACM conference on computer-supported cooperative work*, pages 273–280, 1992.

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

[26] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Technical Conference*, pages 183–195. USENIX, June 1994.

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

[28] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

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

[30] 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.

[31] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE symposium on security and privacy*, 2002.

[32] Jim Snow. Auditing filesystem activity at the page cache layer. `http://syn.cs.pdx.edu/wiki/index.php/Forensix`.

[33] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Security Symposium*, pages 185–198, August 2000.

[34] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[35] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In

Proceedings of the Operating Systems Design and Implementation (OSDI), pages 165–180, 2000.

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

[37] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*, pages 172–183, December 1995.

[38] Ken Thompson. Reflections on trusting trust. *Communication of the ACM*, 27, August 1984.

[39] Andy Watson and Paul Benn. Multiprotocol Data Access: NFS, CIFS, and HTTP. Technical Report TR3014, Network Appliance, Inc., 1999. `http://www.netapp.com/tech_library/3014.html`.

[40] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 17–31, 2002.

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

[42] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.