FORENSIC FRAMEWORK FOR HONEYPOT ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Kevin D. Fairbanks

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2010

FORENSIC FRAMEWORK FOR HONEYPOT ANALYSIS

Approved by:

Professor Henry L. Owen, III, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Chuanyi Ji
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor John A. Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Jonathon T. Giffin
College of Computing
Georgia Institute of Technology

Professor Raheem A. Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: March 17th, 2010

For my grandfather, Willie Lee Fairbanks.

You are missed and remembered.

ACKNOWLEDGEMENTS

I have caught enough breaks and had too many of the right people injected in my life throughout the years such that I have garnered a great amount of suspicion about whether or not I have been hurtling through a series of coincidences. I therefore must thank God as I believe that I have been abundantly blessed and without the belief in a higher power, my journeys through the land of academia would have been more harsh and dark than they needed to be.

Without the emotional and financial support of my mother and father, Sharron and Whayman Fairbanks Sr, my pursuit of an advanced degree would have either ended prematurely or have been extensively protracted. I must also thank my bother and sister, Whayman Jr and Veronica Fairbanks, for being my biggest cheerleaders throughout this process.

While in graduate school I have had the opportunity to overhear several stories of bad advisors. This experience has made me appreciate Dr. Henry L. Owen III even more. I can honestly say, that I have never heard anyone speak ill of Dr. Owen and after being under his guidance the last 5 years I can understand why.

I would also like to thank my thesis committee members. Dr. John Copeland and Dr. Raheem Beyah have always been kind enough to share their time if I needed advice when Dr. Owen was unavailable. I really appreciate their input on my thesis. I thank Dr. Jonathan Giffin and Dr. Chuanyi Ji for agreeing to serve on my committee.

The Network Security and Architecture lab members have served as a sounding board many days so I would like to thank Michael Nowatkowski and Joseph Benin for lending me their ears. I also feel a need to acknowledge Dr. Julian Grizzard and Dr. Ying Xia for collaborating with and helping me during the initial stages of my research.

From the Communications Systems Center I must thank many people, but first I need to thank Dr. Christopher P. Lee. He mentored me as I was starting my journey and really

showed me the ropes. I also value my interactions with Dr. Cherita Corbett, Dr. Yusun Chang, Dr. Bongkyoung Kwon, Selçuk Uluagac and Myounghwan Lee. All of the coffee talks and lab administrative tasks helped in ways that I cannot begin to explain.

A key aspect of building many things is first the creation of a strong foundation. Along that notion, I would like to thank my undergraduate mentor, Dr. Didar Sohi, for convincing me to pursue a graduate degree. I would also like to thank Delano Billingsley and Calvin King for keeping me both motivated and grounded throughout our transitions from undergraduate to graduate students and finally into professional life.

I would also like to thank the DHS Fellowship, FACES, STEP, and ORS programs for providing me with financial support while also providing experiences that have helped me become a better person.

I cannot begin to enumerate everyone who has helped me out over the years, but I would like to thank the following people in completely random order: Dr. Gary May, Dr. Bonnie Ferri, Dr. Sekou Remy, Ashley Johnson, Nicholas G. Brown, Dr. George Riley, Beverly Scheerer, Lonnie Parker, Douglas Brooks, Tasha Torrence, Adora Okwo, Jill Auerbach, Julie Ridings, Dr. Donna Llewellyn, Dr. Marion Usselman, the members of the Black Graduate Student Association, my past and present mentees (Sean Sanders, Sahitya Jampana, Kishore Atreya, Kevin Martin, Gellar Bedoya), as well as anyone else who has helped me while I was at Georgia Tech but is not named above.

Last, but certainly not least, I would like to thank my fiance, Karen D. Goodwin, for standing by my side and being patient with me throughout the whole ordeal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this research is to evaluate and develop new forensic techniques for use in honeynet environments, in an effort to address areas where anti-forensic techniques defeat current forensic methods. The fields of Computer and Network Security have expanded with time to become inclusive of many complex ideas and algorithms. With ease, a student of these fields can fall into the thought pattern of preventive measures as the only major thrust of the topics. It is equally important to be able to determine the cause of a security breach. Thus, the field of Computer Forensics has grown. In this field, there exist toolkits and methods that are used to forensically analyze production and honeypot systems. To counter the toolkits, anti-forensic techniques have been developed. Honeypots and production systems have several intrinsic differences. These differences can be exploited to produce honeypot data sources that are not currently available from production systems. This research seeks to examine possible honeypot data sources and cultivate novel methods to combat anti-forensic techniques.

In this document, three parts of a forensic framework are presented which were developed specifically for honeypot and honeynet environments. The first, TimeKeeper, is an inode preservation methodology which utilizes the Ext3 journal. This is followed with an examination of dentry logging which is primarily used to map inode numbers to filenames in Ext3. The final component presented is the initial research behind a toolkit for the examination of the recently deployed Ext4 file system. Each respective chapter includes the necessary background information and an examination of related work as well as the architecture, design, conceptual prototyping, and results from testing each major framework component.

CHAPTER I

INTRODUCTION

1.1 Computer Forensics

1.1.1 The Importance of Computer Forensics

To state that computers are ubiquitous in modern society is an understatement of monstrous proportions. There are many occasions in daily life where a typical individual encounters computing devices. These occurrences include encounters with automated teller machines, modern cash registers, gas pumps, desktops, laptops, and the myriad of servers across intranets and the Internet. What also must be accounted for is the increasing number of digital devices that consumers either ignorantly or knowingly interact with on a regular basis. These devices cover a range from those that may be more apparent such as digital cameras, music players, cellular phones and GPS systems to those which are more obscure due to a computer being included in more complex machines such as automobiles, heating systems, and in some cases household appliances. The existence of modern society in an environment immersed with computers almost guarantees that when an unexpected event happens, be it a crime or failure of a some sort, a digital device may be present recording some form of data. It is for this reason that the field of digital forensics is becoming increasingly important.

If computer forensics is defined as a subset of digital forensics specifically dealing with general purpose machines such as personal computers, Macs, and servers; then it can be seen that there is a need for investigating the events that transpire on these platforms as various forms of nefarious software exist which subvert a user's security. However, the process does face one serious limitation as stated in [8]. Even if the data from a device can be retrieved, computer forensics cannot definitively state who placed the data on the device. In fact there exists methods, discussed further in section 1.1.4, which can prevent data from ever being stored to a computer. In [8], Caloyannides details these methods and goes so far

as to state that computer forensics is really only useful in the apprehension of "small-time crooks or to harass political dissenters." His basic premise is that only the technically inept perpetrators will be caught while the more organized and possibly dangerous targets will be unaffected as they are already using counterforensics.

This viewpoint is countered by Carrier in [11]. In that article, he concedes some points to Caloyannides in stating that it is true that computer forensics will not work if no data is left behind on the storage media. However, Carrier goes on to state that it is acceptable to not create a trail of data for privacy concerns. Furthermore, the list of ways to defeat computer forensics provided in [8] demonstrates the great lengths one must go to in order to cover their tracks. This means that an attacker could miss areas leaving behind other evidence that a forensic analyst could use. To the point of catching less valuable targets, Carrier draws a parallel to that of real-world investigations catching low-level criminals. Although the target may not be a mastermind, it is not acceptable to ignore the threat that they pose. From this standpoint digital forensics is seen as way to obtain data from a system and determine what may have transpired on that system. The techniques used should draw from multiple data sources to reduce doubts about the creator of the data. For example, if a file is saved in several places and backed up to other digital media such as USB drives or recordable CDs; it becomes difficult for the primary user of that system to use the "Trojan defense." (That the computer was compromised somehow and they did not create the data.)

In [43], Schneider states that seeking perfection from software systems is a lofty goal that technology is currently unable to achieve. However, this lack of perfection can be supplemented by adding accountability. Thus audit logs and computer forensics are vital when it comes to determining the at-fault party. Even though digital forensics has some limitations, there are many cases where it has helped solve real-life crimes. The need and importance of this field will continue to grow as computer systems become even more ubiquitous and the general public continues to store personal and professional information not only on traditional computers but portable digital devices as well.

1.1.2 The Forensic Model

Scientifically determining what events have transpired is the primary purpose of performing a forensic investigation, but how this process is undertaken has been the subject of much discussion. In [40], several models were examined in an attempt to develop a generalized digital forensic model. The model developed draws both from the framework of the Digital Forensics Research Workshop (DFRWS) and protocols followed by the Federal Bureau of Investigation (FBI). As such, the final version includes steps that are not strictly technical, since portions of the digital forensic process are an abstraction of practices used in physical or real life investigations. Their proposed investigation model is detailed below:

1. Identification – detection of the occurrence of an incident prompting an investigation.
2. Preparation – collection of the articles and approvals necessary to perform the investigation.
3. Approach Strategy – formulation of a strategy that maximizes evidence collection while reducing the effect an investigation has on the victim.
4. Preservation – protection of the device in question from further tainting or destruction of evidence.
5. Collection – documentation of the scene and duplication of the device.
6. Examination – finding evidence on the device while maintaining detailed logs. This step requires technical knowledge of the device in question.
7. Analysis – development of theories based on the evidence and determining the significance of certain pieces of data. This step does not require the technical knowledge possessed in the previous step.
8. Presentation – summarization of the conclusions with explanations written in non-technical terminology that reference specific details.
9. Evidence Return – return of property to owner after it has been cleared for release.

In this model, the steps that are primarily followed by forensic investigators are 1-2 and 5-8. Peisert et al. examine several different common forensic techniques and tools in [37]. This leads the authors to develop the following principles which they state should be followed in a computer forensic investigation:

1. Consider the entire system.
2. Assumptions about expected failures, attacks, and attackers should not control what is logged.
3. Consider the effects of events, not just the actions that caused them, and how those effects may be altered by context and environment.
4. Context assists in interpreting and understanding the meaning of an event.
5. Every action and result must be processed and presented in a way that can be analyzed and understood by a human forensic analyst.[37]

The first principle requires that a system designed to perform forensic analysis or collect forensic evidence have the ability to access multiple layers of abstraction. For memory analysis, the system should not only consider events that have transpired in kernel space, but what has taken place in user space as well. This principle also extends to file systems as the authors state forensic methods that only consider file system events while not capturing actions that write directly to a device can be subverted.

The second principle is aimed more at network and system administrators than at the developers of forensic tools. It is here that improper identification of potential threats could lead to a lack of forensic evidence for post attack analysis. For example, if it is assumed that the system will be attacked remotely, then mechanisms may not be set in place to handle improper actions by local users. It is not enough to depend on the default logging mechanisms or one particular evidence collection tool, administrators must use some combination of tools and not allow prior assumptions to taint a current investigation.

Principles three and four relate to context which the authors define as the environment in which a program executes including its input arguments. Depending on how an attacker

modifies his environment, an action can have a multitude of results. This requires an investigator to not only have the ability to track keystrokes, but also detect environmental changes. The last principle is about information presentation. The importance of information presentation cannot be stressed enough. Even if it were possible to log every event on a system in a scalable manner, unless the data is presented in a way that an investigator can properly interpret what has transpired, it may be useless.

1.1.3 Forensic Tools and Techniques

There exist many open source and commercial tools and techniques developed for conducting forensic investigations. Among forensic toolkits, The Coroner's Toolkit (TCT) [22] [23] can be thought of as the starting point which has been improved upon by The Sleuth Kit (TSK) and its graphical user interface the Autopsy Browser [14]. As a matter of course, the Autopsy Browser has been improved upon through the development of PTK[1] [16]. These open source tools are mirrored by commercial products such as EnCase by Guidance Software [26] and the Forensic Toolkit (FTK) by AccessData [1].

Although most of the aforementioned sets of tools have been well tested and enjoy widespread use in the forensic community, they are mostly used in what is known as dead forensics. Dead, static, or quiescent forensics is the practice of analyzing a system after it has been halted. This usually involves literally pulling the plug on a device to prevent any logs or other files from being further altered during the regular shutdown process. In [27], some of the positive and negative aspects to this approach are summarized. The conclusion drawn is that live or non-quiescent analysis is the next logical step in the evolution of the digital forensic practice. To that end, in the same document a brief survey of the current live analysis techniques is presented.

As stated in [27], static forensics is the most mature type of analysis; however, it also presents the examiner with only a partial view of a target system's state. The analysis of a file system will not likely reveal information such as the names and number of the processes that were running, information about network connections, as well as other memory resident

[1]No definition for "PTK" was readily available at the time of this writing.

data such as encryption keys. This information can be vital due to the fact that encryption schemes for partitions and entire disk drives are becoming more common. Thus live analysis techniques are being researched and developed such that a more complete picture of what has transpired can be obtained.

In [27], Hay et. al. list several types of live analysis which include:

- Directly accessing a target system
- Using trusted imported binaries
- Software modification for monitoring
- Hardware modification for monitoring
- Live virtual machine analysis
- Analysis through virtual machine logging and replay

Each of these categories has strengths and weaknesses which range from the limited applicability of virtual machine techniques to problems with the integrity of the system being analyzed. One major problem with several of the techniques listed above is that while investigating the target system, the analyst is also altering the state of the memory on the system. This means that the results of a query cannot be reproduced in the same way that they are in static analysis. Although repeating the steps performed in the examination of a memory dump will probably yield the same results, a result of operating in the dynamic environment of a live system is that running the same command twice (i.e., to obtain the memory dump) can yield different results.

The last two categories warrant further discussion as the use of virtual machines as a subject of forensic investigations has and will continue to increase. Virtual environments provide several benefits such as the rapid deployment and reproducibility of entire systems. For example, if a company is running a server in a virtual environment, it may be easier to redeploy a clean copy of the virtual machine instead of troubleshooting a potential problem. For that matter, it may be even more advantageous to do both tasks in parallel. Because

the virtual machine runs on top of a physical environment, it introduces another security profile as an attacker must compromise the virtual machine and possibly the virtual machine monitor to gain complete control of physical resources. The focus now becomes securing the virtual machine. As a virtual machine can be paused, virtual machine introspection provides viewpoints into VMs that are difficult to achieve in real machines such as detailed memory analysis and the tapping of I/O devices. VMware's VMsafe and the XenAccess library, are examples of projects and products that can be extended to build more complicated introspection tools. For example, the Virtual Introspection for Xen (VIX) project has yeilded several command-line tools. Still, as pointed out in [27], obtaining raw memory dumps and interpreting them are two separate tasks. Furthuremore, VMs are becoming more common, but they are not everywhere. Other benefits and drawbacks to virtual machine introspection are discussed in [37].

1.1.4 Anti-Forensics

The state of computer forensics has shifted from a period of discovery to an arms race. Just as techniques have been developed to obtain forensic evidence from current technology, anti-forensic techniques have been employed to obfuscate an attacker's actions. In [23], the authors note how modification, access, and inode change timestamps for files can be used to assemble a timeline of events which have transpired on a compromised system. This publication also mentions how vulnerable these timestamps are to tampering. In Linux, utilities such as `touch` can be used to change this metadata and a tool known as `TimeStomp` is available to manipulate metadata times in NTFS [35]. Garfinkel discusses several anti-forensic techniques in [24] which include but are not limited to the manipulation of data and metadata, data hiding, cryptographic techniques and a number of other observed behaviors. Anti-forensics is further discussed with comments from several forensic professionals in [4], while [20] specifically discusses techniques for data hiding.

In [8], Caloyannides lists several counter forensic techniques. They range from physical considerations such as tilting monitors and using of shrouds to avoid cameras to more technical aspects like reducing the risk of the interception of Van Eck radiation, encrypting

network traffic, and using trusted proxies to anonymize IP addresses. Clearly, the type of anti-forensics a person can employ is highly dependent upon the amount of control they have over their environment. One thing that Caloyannides points out in his discussion is that since there are multiple techniques used to collect evidence, several anti-forensic techniques must be combined to balance the risk of a forensic attack.

1.1.5 Legal Considerations

As the target systems in this thesis are honeypots, the legal aspects of computer forensics are not a focus of the research which is presented in the subsequent chapters. However, a brief overview of some of the more important issues is provided here for the sake of completeness. As summarized in [33], the rules for admissibility of digital evidence in court are influenced by the Federal Rules of Evidence (FRE), the Daubert standards, and case law. These three sources do not make up an iron clad set of standards, therefore rules can vary from state to state. However, as more states begin to align themselves with federal practices, these (especially the FRE) should be considered sources for best practices. Other issues include the licensing and certification of forensic examiners.

There exists no nationally recognized license for digital investigators. In most states, the analyst must obtain a license as a private investigator. Although this may seem simple enough, it is complicated by the fact that the license from one state is probably not acceptable in a different state. Also, the agencies that administer the licenses vary from state to state. It is pointed out by Manes and Dowing that this situation is similar to how the medical profession is licensed. Licensing is a necessary evil as the organizations which administer the licenses can penalize and fine individuals who do not follow proper procedures. The penalties are not limited to just the analyst but can extend to the hiring attorney in some instances.

As with licenses, there is no one major certification for forensic analysts. According to [33], this has been the topic of much debate since the mid 1990s. Currently the landscape is littered with certifications provided by vendors, trade schools, and degrees from various institutes. The problem that arises is that no consistent set of core knowledge or skills

is guaranteed in this environment. Therefore customers must consider reputation, experience as well as academic qualifications. As the field is relatively young and the laws and requirements regarding digital evidence are literally being shaped case by case, it is the responsibility of the examiner to familiarize themselves with the appropriate laws of each state in which they collect and/or process data.

1.2 Production Systems and Honeypots

Overall, existing forensics toolkits are used to determine the transactions that have taken place on a production, or production-like, system. They offer many utilities that aid in accessing the multiple layers of abstraction to retrieve data. They can also greatly reduce the labor required when examining many logs by gathering and presenting data in a way that is useful to investigators. These toolkits, however, are made to analyze production systems.

One major difference between honeypots and production systems is that honeypots serve as reconnaissance mechanisms while the function of a production system is user defined. This division in usage leads to the realization that resource consumption for the two types of systems can also differ, giving a honeypot the leeway to use a greater percentage of its resources toward security monitoring than would be practical for production systems.

While a production system may be utilized by one or more parties and usually stores some set of data that may be important, a honeypot has no actual users nor does it contain important production data. After an attack, data recovery may be attempted if a production system is compromised, but on a honeypot the hard disk could be taken out for analysis, possibly stored, and then eventually formatted and reused. Data existing before an attack is inconsequential.

In a honeynet environment, all network activity produced by a honeypot is monitored and stored as any traffic, especially inbound, is suspicious. Traditional forms of information such as packet captures and flow logs can be stored for a relatively long period of time depending on the amount of attack traffic. Production systems differ in that regular user activity can possibly generate far more network traffic over the same amount of time than

a honeypot. This massive volume of data limits the amount of state that a production environment can keep.

In general, resource usage on production systems is geared toward increasing the performance of that system for some application. This need not be the case for honeypots. Although honeypots need to mimic production systems to entice attackers, more resources can be geared toward the monitoring of the honeypot than that of the production system. With that in mind, more processor cycles and memory can be used to log incremental events such as the tracking of changes in file metadata to aid in the forensic analysis of attacks.

1.3 Statement of Research

Existing forensic methodologies are very useful when attempting to determine the transactions that have taken place on an enterprise production system. They offer many techniques that aid in accessing multiple layers of abstraction to retrieve data. They also ease the pain of going through a multitude of logs. These methodologies, however, are made to analyze enterprise production systems. Enterprise production systems are characterized by having very limited resources to dedicate purely to security. Therefore, the persistence of journals and log files vary depending on hardware and software implementations and limitations. Honeypots differ in that they offer the advantage of not having to provide any enterprise network or database production value.

Honeypots and enterprise production systems are two fundamentally different types of systems with completely unrelated purposes. For example, while there are many valid transactions that take place on enterprise production systems, any transaction that takes place on a honeypot is subject to suspicion. Because of these differences, many assumptions made about enterprise production systems do not hold true for honeypots. There is no universally applicable methodology for the forensic analysis of all types of systems; this creates an opportunity to research honeynet forensic methodologies that take advantage of the unique honeypot characteristics. This research investigates methodologies for enhancing honeypot forensics.

The hypothesis investigated in this research is that with forensics enhancements built

directly into the operating systems, additional data may be harvested which will increase the probability of a successful forensic examination of a honeypot. The research can be separated into the three distinct areas summarized below.

TimeKeeper

In the field of Computer Forensics, gathering file metadata times such as the modification, access, and inode change times (MAC-times) can be very useful. This source of information can be used to build a timeline of the events which have transpired on a system. One problem with relying on this as a primary source of information is that an attacker can alter these times. This research seeks to increase the robustness of MAC-times by investigating a methodology for preserving them. In particular, the Ext3 journal was examined and used as a source of information for gathering the timestamps soon after they have been updated.

Dentry Logging

While investigating timestamp preservation, it was observed that the basic unit of work in the Ext3 journal was a file system block. Although this is very useful when processing timestamp information as locality can be exploited when examining inode structures, it presents a problem when attempting to perform an inode number to filename translation. This is primarly due to fact that the design of the Ext3 filesystem uses one way pointers to translate from a filename to a particular inode number. This research investigates a method to enable inode number to filename translation by taking advantage of the Virtual Filesystem (VFS) Layer in the Linux kernel as well as a historical cache of MAC-time information.

Ext4

Although Ext3 has traditionally been the default file system for many Linux distributions, it has certain limitations in terms of scalability that resulted in significant changes when compared to Ext4. This has resulted in Ext4 becoming the default file system for several major Linux distributions and created a need for the forensic community to have

methods for the examination of file systems in this format. In this research, Ext4 is thoroughly examined and it is demonstrated how methods used to analyze Ext3 can result in inaccurate information when performed on an Ext4 partition. It is further shown that due to preallocation, there exists a danger of data persistence within sparse files.

CHAPTER II

TIMEKEEPER

Production systems, when compared to honeypots, are characterized by having very limited resources to dedicate purely to security. Therefore, the persistence of journals and log files vary depending on the hardware and software implementations and/or limitations of a particular system. Honeypots differ in that they offer the advantage of not having typical production system constraints. These differences can be leveraged to create a richer set of data for honeypot forensics than what is capable for production systems. This chapter presents the concept of a honeypot targeted journal monitoring methodology named TimeKeeper [21].

2.1 Background

Although Computer Forensic techniques can be generalized into the steps detailed in Section 1.1.2 that are independent of a particular file system, each file system is implemented with a particular goal or set of goals in mind. Therefore, each file system presents its own unique set of problems and opportunities. This means that understanding the details of the file system in question is a key component in gathering forensic evidence. This is directly reflected in step 6 of the forensic model presented in Section 1.1.2.

When undertaking a digital forensic analysis on a file system, a key step is the development of a timeline to discern when certain events took place. Obtaining the modification, access, and change times of a file can be valuable in this process [23]. When examined with a trained eye, this data source can not only reveal intrinsic information about a file, but also give details about intrusions. However, this data source is temporary and there exist anti-forensic techniques to corrupt this information [24]. As a response to these techniques, a methodology has been investigated to track file metadata times in honeypots to produce a reliable source of information for forensic analysis.

The Third Extended File System (Ext3) was a prime candidate for this research as

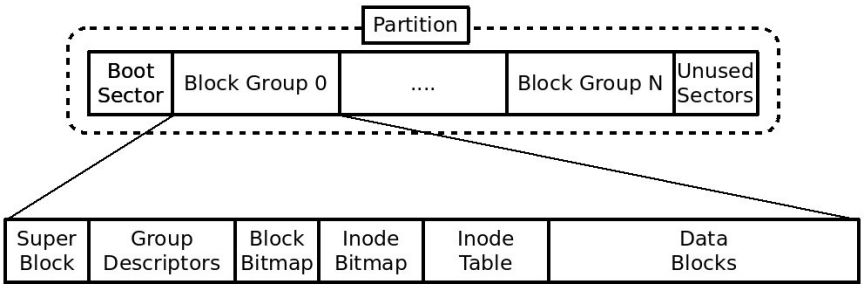

Figure 1: File System Layout [2]

it is open source and was the default option for most major Linux Distributions[1]. In addition, due to its compatibility with the Second Extended File System (Ext2) while adding journaling capabilities, the file system has amassed a significant following.

This chapter presents a methodology which has been investigated and deployed that successfully tracks modification, access, and change times in a way that makes malicious modification of timestamps originating from the target file system difficult to hide. The benefit of this approach is that it makes forensics using metadata times more reliable and useful in honeynets. The metadata time information is harvested from the Ext3 journal. This design choice allows for the identification of suspicious file modifications that attackers may attempt to conceal through the altering of metadata. It can also serve as a starting point for a full forensic investigation, or yield more insight into an ongoing one. With these circumstances in mind, first a discussion of the Ext3 file system is provided as background before presenting the methodology and results.

2.1.1 ExtX

In [13], Brian Carrier discusses Ext2 and Ext3 together using the term ExtX because Ext3 can be thought of as an extension of Ext2. Both file systems share the same basic data structures and a key factor in the development of Ext3 is its backward compatibility with the already popular Ext2. For these reasons, this discussion serves as an introduction to both file systems with any key differences between them further explained.

The basic unit of work for most hard disks is the sector which is usually 512 bytes

[1]This research was performed before the adoption of Ext4, but the concepts presented are still valid.
[2]Adapted from [5]

in size. File systems, on the other hand, tend to operate on units that are comprised of multiple sectors. In ExtX, these are called blocks. ExtX supports 1024, 2048, and 4096 byte block sizes. As displayed in Figure 1, these blocks are organized into units appropriately called block groups. Each block group, with the possible exception of the last, has an equal number of data blocks and inodes. An inode stores basic metadata information about a file such as timestamps, user and group permissions, as well as pointers to data blocks. As every file must have a unique inode, the number of files that a system can support is limited by the total number of inodes. In ExtX, not all blocks are equal. Along with data blocks, the file system uses group descriptors, inode and data block bitmaps, inode tables, and a file system super block.

The ExtX super block is a metadata repository for the entire file system. It contains information such as the total number of blocks and inodes in the file system, the number of blocks per block group, the number of available blocks in the file system, as well as the first inode available. A group descriptor contains meta information about a particular block group such as the range of inodes in the group, the blocks included in the group, and the offset of key blocks into the group. Originally, the super block and group descriptors were replicated in every block group with those located in block group 0 designated as the primary copies. This is no longer common practice due to the Sparse Super Block Option [12]. The Sparse Super Block Option only replicates the file system super block and group descriptors in a fraction of the block groups. This can allow for a significant amount of space savings on larger partitions that can have a multitude of block groups and therefore a larger set of group descriptors.

Each group has an inode bitmap and a data block bitmap that is limited to the size of one block. These bitmap blocks limit the number of inodes and data blocks of a particular group. As the maximum block size currently supported in ExtX is 4096 bytes, the maximum number of inodes or blocks in a group is 32k. In the bitmaps, a 1 is used to identify an unavailable object and a 0 denotes that the object is unused. If the number of objects that a group can theoretically contain is greater than the number that it actually contains, the corresponding bitmap spaces are set to 1.

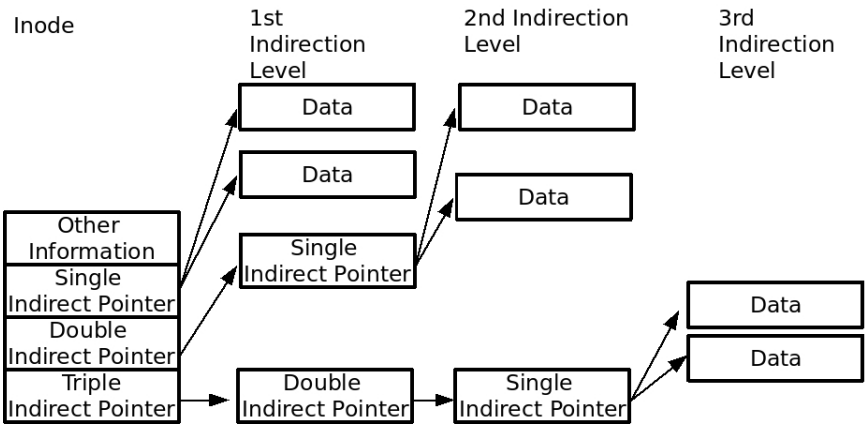

Figure 2: Inode Data Block Pointers [3]

In ExtX an inode is 128 bytes in size. Inodes are stored sequentially in a structure called the inode table that is present in each group. The size of the inode table is related to the block size the file system uses. For example, using a 4096 byte block, the maximum number of inodes per a group is 32k inodes. As each block can hold 32 inodes, the maximum size of the inode table in this example would be 1024 blocks.

In Figure 2, the almost ubiquitous diagram portraying the inode data block pointer system can be seen. In an effort to keep the inode structure from becoming too large while giving it the ability to support large file sizes, different types of pointers were implemented. The most straightforward method uses single indirect pointers which are stored inside of the inode structure. These point directly to file data. The double indirect pointer, on the other hand, points to a file system block which acts as a repository for single indirect pointers. The triple indirect pointer adds another level of indirection onto this scheme. More advanced file systems, such as Ext4 in section 2.1.3, may use structures known as extents to denote sections of contiguous data blocks, but Ext3 uses the pointer system described above for compatibility reasons [19].

The primary intention of this detailed explanation of the pointer system is to bring attention to how Ext2 and Ext3 handle inode structures differently. Upon file deletion, Ext2 marks the inode associated with the deleted file as available by editing its inode and data block bitmaps. This practice makes it possible to recover a deleted file if its contents

[3] Adapted from [13]

have not been overwritten, as the inode structure will contain pointers to the data blocks. Ext3, in contrast, performs the additional step of overwriting block pointers in the inode structure upon deletion. This increases the difficulty of file recovery and requires the use of the technique known as data carving, further explained in section 2.2, to reconstruct the file. A more complete description of ExtX can be found in [13] and [5].

2.1.2 Journaling

Journaling is a technique employed by file systems in crash recovery situations. This method offers reduced recovery time at the cost of additional file system overhead in terms of time and/or space. The journal, in Ext3, takes on the form of a fixed size log that regularly overwrites itself. Actions are recorded to the journal and either replayed in their entirety or not at all during file system recovery. This feature, called atomicity, allows Ext3 to come to a consistent state more quickly than through the use of a file system check program such as `e2fsck`.

With Ext3, the actual journaling action is handled separately from the file system by the Journal Block Device. The Ext3 file system is used to identify transactions which are then passed to the Journal Block Device to be recorded. A transaction is a set of updates sent to the device which are uniquely identified by sequence numbers. The transaction sequence number is used in descriptor and commit blocks to denote the start and end of a transaction respectively. The data that is being journaled is nested between the descriptor and commit blocks. The descriptor block also contains a list of the file system blocks that are updated during a transaction. It is important to note that the commit block is only created once the file system data has been recorded to the disk to ensure atomicity.

There also exists a revoke block. This, like the descriptor prevent, has a sequence number and a list of file system blocks, but its purpose is to block changes during the recovery process. Any file system block listed in the revoke block, that is also in a non-committed transaction with a sequence number less than that of the revoke block is not restored.

The next structure that the journal uses is probably one of the most important: the

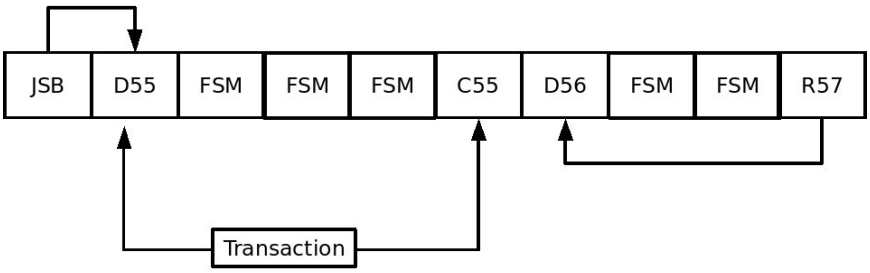

Figure 3: Ext3 Journal Blocks [3]

Figure 3 Key

Abbreviation	Meaning
JSB	Journal Super Block
D[X]	Descriptor Block X
FSM	File System Metadata
C[X]	Commit Block X
R[X]	Revoke Block X

superblock. This should not be confused with the file system superblock, as this is specific to the journal. This structure has the task of identifying the first descriptor block in the journal. The usage of the journal as a circular log makes this task necessary, as it cannot be assumed that the beginning of the journal is located at the beginning of its physical storage space [13]. This relationship between the different types of journal blocks is displayed in Figure 3. In this figure, transaction 55 is complete as the set of metadata blocks associated with it are enclosed with a descriptor block and commit block. However, transaction 56 was not completed so no commit block was created. When the file system is mounted again and the journal checked, a revoke block will be created. Since transaction 56 has no commit block and it has a sequence number less than 57, the changes associated with this transaction will not be replayed.

The three documented modes of journaling are Journal, Ordered, and Writeback [5]. Journal mode, while certainly being the safest method, is the most costly of the three in terms of performance. It writes both file data and metadata to the journal and then copies the information to the actual file system area on the disk. To avoid writing everything twice, Ordered mode only writes metadata to the journal. In this mode, care is taken to write to the file system disk area before writing to the journal area. In most Linux distributions, it

is the default mode in which the journal operates. In Writeback mode, only the metadata is journaled, but no precaution is followed to insure data is written to the file system area first.

The difference between the Ordered and Writeback modes is a tradeoff between performance and safety. If the data is written to disk first but the metadata is not journaled before a system crash, then the system will come to a consistent state under the assumption that the last transaction did not take place. If the crash occurs while in the process of writing to the journal then the transaction will be revoked. Writeback mode can result in writes to the journal area and the actual disk blocks being interleaved. If this is the case, upon a system crash, neither the journal nor the disk blocks will be consistent resulting in the file system being left in a corrupt state.

The modes of journaling are a compromise between the impact a journal has on system performance and the ability that the mode has to recover in reaction to an unclean unmounting of the file system. These modes also have an effect on the amount and types of forensic evidence that can be gathered from the journal as well as the rate with which the journal overwrites itself.

2.1.3 Ext4

Although most of the research presented in this document was designed with the Ext3 file system in mind, the recent adoption of Ext4 as the default file system for several major Linux distributions makes a discussion of the differences between it and the preceding versions of ExtX necessary. In 2006, a series of patches to extend the capabilities of Ext3 was developed; however, complete compatibiltiy could not be maintained as on-disk structures needed to be changed resulting in a branch from the stable code. Thus, Ext4 can be best thought of as an evolution of Ext3 rather than a mere extension. It leverages the previous knowledge gained through the development of Ext3 while adding the benefits of more recent file system management techniques. The releases of Fedora 11 [45], Ubuntu 9.10 [9], and OpenSuse 11.2 [36] in June, October, and November of 2009 respectively, have Ext4 as the default file system choice for new installations. This move ensures that forensic investigators will have

to analyze hard disks formated with the new file system. Also, as honeypots must resemble production systems, having Ext4 installed on honeypots is an inevitable occurence. More detail about the file system and the forensic implications of certain changes are included in chapter 4.

2.1.3.1 Topology

Although Ext4 has the same basic data structures as its predecessors, there are differences. Some of the more in-depth differences will be covered below, but one major difference that should be immediately addressed here is the organization of the block groups. In [34], it is stated that the 128MB block group size limit has the effect of limiting the entire file system size. This is due to the fact that only a limited number of group descriptors can fit into the span of one block group. In the literature, there have been two solutions mentioned for this problem: the meta-block group feature and the flex block group feature.

The meta-block group feature was first mentioned in [52]. There it is stated that a meta-block group consists of a series of block groups that can be described by a single descriptor block. More detail is given in [34] as backups of the meta-block group descriptors are said to be located in the second and last group of each meta-block group.

Figure 4 graphically demonstrates how the flex block group feature extends the previous notion of creating large contiguous swaths of block groups by moving the block and inode bitmaps as well as the inode tables to the first block group in a flex block group along with the group descriptors [30]. With the sparse superblock feature being enabled by default some block groups may contain backup copies of the superblock. This centralization of metadata blocks from different groups creates a virtual block group which allows free contiguous spans of blocks to be allocated across group boundaries. On the development mail list archives, there can be observed discussions about whether the two options should have different feature flags. It was agreed that this was necessary so that a clear definition of META_BG and FLEX_BG is maintained. In the end, it seems that the FLEX_BG feature is the most important to understand as it is selected by default in the mke2fs.conf file.

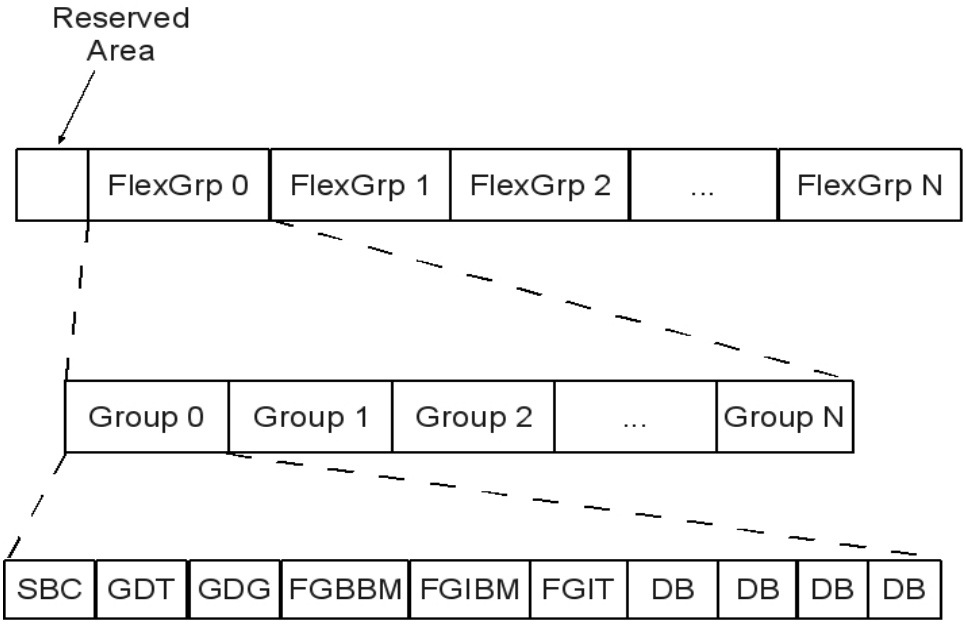

Figure 4: Flex Block Group Layout

Figure 4 Key

Abbreviation	Meaning
SBC	Super Block Copy
GDT	Group Descriptor Table
GDG	Group Descriptor Growth Blocks
FGBBM	Flex Group Block Bitmap
FGIBM	Flex Group Inode Bitmap
FGIT	Flex Group Inode Table
DB	Datablock

2.1.3.2 Scalability

Setting aside the physical topology of the file system, one of the most noticeable features that Ext4 introduces to users over Ext3 is its ability to support very large file systems. Ext3 is limited to maximum of file system size of 16 terabytes (TB), because there are only 32 bits available to address data blocks which have a 4 kilobyte (KB) default size. This limit is exceeded by increasing the block number space to 48 bits. When combined with default block size, this yields a maximum file system size of 1 exabyte[4]. This change in the block number field has caused new fields to be introduced in the file system super block

[4] $1 exabyte = 1000 petabytes = 10^6 terabytes = 10^9 gigabytes = 10^{18} bytes$

and block group descriptor structures so that they can accommodate values up to 64 bits while maintaining backward compatibility with the preexisting 32-bit structures.

Along with a drastic increase in maximum file system size, the maximum individual file size has been increased. While in Ext3 the maximum size of a file is 2 TB, an Ext4 file reaches its limit at 16 TB. This increase is acheived by adding a flag (HUGE_FILE) to the file system and in some of the inodes to denote that the files they represent are very large. When these flags are set, the 32 bit field that is normally interpreted as the number of 512 byte sectors, is instead interpreted as the number of file system blocks.

Another major feature in Ext4 is the use of extents rather than the indirect block mapping method shown in Figure 2. Extents are more efficient at mapping data blocks of large contiguous files as their structure generally consists of the address of the first physical data block followed by a length. In Ext4, an unbroken span of up to 2^{15} blocks can be summarized with one extent entry. If the block size is 4KB, this is up to 128 MB of data. In [34], it is stated that an inode can hold four extents. If a file is highly fragmented, very large, or sparse then a tree of extent structures is constructed.

In the previous examinations of Figure 1 and Figure 4, one classification of blocks was not discussed. These are the blocks reserved for the growth of the group descriptor table. These blocks help to facilitate file system expansion as described in [10]. When the file system is being expanded, and a new block group must be added, a new group descriptor must be added accordingly. If there is no extra room at the end of the last group descriptor block a new block can be allocated from the reserved area. This allows a file system to grow over time rather than keeping it at a fixed size as would be the case in an Ext2 or Ext3 formated partition.

Another scaling problem with Ext3 is the 32000 limit on the number of subdirectories which can be created. Ext4 has no subdirectory limit. When the link counter of a directory overflows its 16bit space, the count is replaced with a 1 to denote that the directory is not empty. The reason huge directories can be supported is due to the new way entries are stored. In Ext2/3 a linked list was used by directories, with the Linux Virtual File System (VFS) caching mechanisms helping ease the associated timing costs. Ext4 breaks with that

tradition by using constant depth hash trees. These H-Trees decrease lookup times, thereby helping to improve performance in large directories. Details about this design choice can be found in [10].

2.1.3.3 Compatibility

One thing to note is that unlike the compatibility between Ext2 and Ext3, Ext4 is only partially compatible with the older file systems. Ext3 partitions can be mounted using Ext4. In this type of hybrid environment, older files will continue to use indirect block mapping while newly created files will use the larger inode structure and extents. It is possible to obtain some of the benefits of Ext4 by mounting an older partition without extent support. However, due to the fact that the on-disk structures used by the older file systems and the new one are different, Ext4 cannot be mounted by Ext3.

2.1.3.4 Block Allocation

Along with changes to improve scalability, the way that blocks are allocated by the file system has been altered. Preallocation, one of the more interesting additions, is the process of reserving blocks for a file before they are actually needed. More importantly, when these blocks are allocated, they are not initialized with data or zeroed out. This is different from the block reservation system proposed in [34] for Ext3. In that scheme, blocks were allocated based on reservations that were stored in memory structures. This means that across reboots the preallocated range could be lost. In Ext4, this is solved through the use of extents. The most significant bit of the extent length field is used to denote a span of blocks that contains uninitialized data. When the these spans are read from, they are interpreted as zero-filled blocks. This allows an expansive collection of contiguous blocks to be saved for a particular file while also saving time as large sparse files do not have to initialize every block they reserve.

Another change that has sparked some debate is delayed allocation. The basic idea behind this feature is that instead of the file system blocks being used one at a time during a write, allocation requests are kept in memory. Blocks are then allocated when a page is

flushed to disk. This saves time as what previously took multiple requests can be summarized with one. Other benefits include not having to allocate blocks for short lived files and a reduction in file fragmentation. In [30], it stated that delayed allocation is handled at the VFS layer.

The Ext4 multiple block allocator is detailed in [30]. It is designed to handle large and small files by using per-inode preallocation and per-CPU locality respectively. The strategy that is used for a particular file depends on a tunable parameter that specifies the number of blocks requested for allocation. The general idea is that smaller files using the same CPU will be placed physically close together and larger files will achieve less fragmentation through preallocation.

2.1.3.5 Reliability

The journaling used with Ext4 is a little different than what was previously described with Ext3. [34] describes a CRC32 checksum being added to the commit block of a transaction. The CRC is computed over all of the transaction blocks. During recovery, if the checksum does not match, then that transaction and all subsequent transactions are treated as invalid. The cost of computing a checksum for a transaction is offset by the fact that commit blocks can now be written to the journal with the rest of the transaction as opposed to the two stage commit process previously used. This can be done because the checksum can detect blocks that are not written in the journal. In [34], it is stated that the end result is the file system speed is actually increased by up to 20%.

Group descriptors are made more robust with the use of a CRC16 checksum. This is used to verify that the unused inode count field in the descriptor is valid when a file system check, done by `e2fsck`, occurs. This unused field along with new flags that denote whether the bitmaps and inode table are uninitialized or not let `e2fsck` skip areas that have not been used thereby reducing the time it takes for the file system to be examined. Furthermore, there exists a `lazy_bg` option for the `mke2fs` command which saves time when creating large file systems by not writing the inode tables at the creation of the file system.

Fragmentation is one of the banes to performance in many file systems. As the file

system ages and free space is depleted, the risk of fragmentation increases. This is especially true for very large files. In [42], several techniques are discussed which combat this trend while the file system is still mounted. The overall goal is reduce single file, relevant file, and free space fragmentation. Generally, this is done by either moving fragmented data to contiguous free space, or making uninterrupted space for larger files by moving the data belonging to smaller files to another area. Online defragmentation patches are available, but support has not been merged into the main line kernel yet [53].

2.1.3.6 Time Related Enhancements

As noted in [56], Ext3 is limited to second resolution at the file system level. This is addressed in Ext4 by modifying and extending the inode structure. As opposed to the default 128 byte inode previously used, newer structures will be 256 bytes. This space allows Ext4 to support nanosecond timestamps. Also, a new timestamp field has been added to document the file creation time. Each timestamp field in the inode has a corresponding high 32 bit field. In this space, only 30 bits are used for nanosecond representation. The 2 remaining bits are used for the extension of the epoch, thereby delaying the 2038 problem for 272 years[5]. This increase in time resolution from the file system perspective may help future computer forensic and security research. All of these changes have affected journaling as the JBD (section 2.1.2) has been branched into JBD2 which can support journaling of 32-bit and 64-bit file systems.

As Ext4 is fairly recent, some features may continue to be modified and added, however, the on-disk data structures have become stable and should remain the same. This file system is more greatly detailed in [30], [34], [42], and [52].

2.2 Related Work

2.2.1 Inotify

Inotify is a file monitoring system developed by Robert Love and John McCutchan that has been integrated into the Linux kernel since version 2.6.13 [32]. It is a replacement for the Dnotify file change monitoring system and offers several advantages over its predecessor. For

[5] January 19th, 2038 at 03:14:07 UTC is the latest time that can be represented by a signed 32 bit number

example, Dnotify was limited to monitoring directories through the use of file descriptors. Inotify, on the other hand, only uses one file descriptor and, as the name implies, is inode based so it can monitor individual files giving it greater resolution to file system changes. Inotify has system calls in the kernel giving it the ability to create an inotify instance and watches. The instance uses a single file descriptor and can be associated with many watches. It is the watch that actually does the work. In the creation of a watch, it is specified with which inotify instance the watch is associated, what file or directory should be monitored, and what events should be reported. Watches can be modified and removed without having to create a new instance of inotify. Currently, notification for the following events is supported:

- content access
- metadata modification
- deletion
- content modification
- creation
- moves

Inotify events can be received by reading them from a buffer, polling, or using the select system call. The latter two options allow for none threaded applications to function in a nonblocking manner. As kernel memory is limited, a restriction is placed on the number of inotify instances and the watches per instance that are allowed. Also, if a great number of events occur in close succession, it is possible for the inotify event queue to fill and drop new events. In this case, an overflow event will be sent. The default values for these limiting parameters are located and can be modified using *procfs* or *sysctl*. While Inotify works by monitoring system calls, TimeKeeper uses the Ext3 journal as a source of information in an effort to offer a more scalable solution. Inotify is discussed in greater detail with usage examples in [32].

2.2.2 Sebek

According to [50] Sebek is the "primary tool to capture attacker activity on high interaction honeypots." As honeypots are "designed to be compromised by an attacker" [47], any activity is subject to scrutiny which results in an incessant state of monitoring. In reaction to the monitoring techniques, attackers have proceeded to conceal their actions through the

use of techniques such as encryption. Sebek's main purpose is to counter this behavior.

In [49], it is noted that honeypots can contain trojaned binary programs to obtain information about an attack. For instance, the user environment can be altered to reveal every command that is entered by an attacker while making it a nontrivial matter to erase any logs produced. This practice has resulted in attackers using tools of their own, which they download after the initial compromise. Moreover, an attacker that has compromised a remote machine may install some form of session encryption software. This will make network activity between the compromised machine and the attacker virtually useless without the session keys.

Sebek comes in the form of a client module and the Sebek server. The client, which is based on a rootkit, is a Linux kernel module which places Sebek in privileged space. Since it operates at kernel level, it has the ability to circumvent encryption schemes by intercepting encrypted data as it is decrypted to be executed. Some of its touted features include keystroke logging and the capture of passwords for encrypted programs such as burneye binaries. To gain this functionality, the Sebek client actually replaces the `read()` system call with its own which logs the interesting data before calling the original `read()` system call. Not wanting to leave the data on a compromised box, the client then exports the data to the Sebek server which resides on a completely different physical machine. It is assumed that most of the attackers who are savvy enough to infiltrate a box remotely are also able to detect network traffic exiting that box. Because of this, the Sebek client also modifies the network protocol stack of the honeypot to conceal any Sebek data through the use of a magic number in Sebek packets. If every honeypot in a honeynet has Sebek installed, then no honeypot on the network would be able to detect Sebek generated traffic [49].

It should be noted that Sebek is neither completely invisible nor invincible. Dornseif et al. have used several techniques to detect earlier versions of the utility which include monitoring network congestion due to heavy Sebek traffic and examining the location of the `read()` system call to detect a modification [17]. They even went as far as to construct *Kebes*, an anti-Sebek toolkit designed for the detection and disabling of the honeypot tool.

Furthermore, there exist several documents detailing the detection of questionable environments such as in [28]. These documents demonstrate the need for continued research in the area of honeypot evidence collection, which is one motivation for research presented in this thesis.

2.2.3 Zeitline

In [7], the design and implementation of a graphical user interface specifically targeted at the creation and editing of timelines is discussed. This open-sourced tool, implemented in Java due to its platform independent nature, is called Zeitline. Unlike lower-level forensic tools such as `mactime` or `fls` which focus on the retrieval of data, Zeitline is geared more toward data analysis. Instead of retrieving data for forensic purposes, it is designed as an aggregation mechanism for the event logs produced by other sources. Furthermore, it displays events in a graphical tree and allows the simple events that are produced by one or more logs to be grouped into what the authors call a complex event.

In order to import events from a variety of sources, Zeitline loads input filters at run time. This allows a more technically savvy user to extend the `InputFilter` interface to match their specific needs. Although this editor has great potential, the need for end users to possess Java knowledge is somewhat limiting. Also, as of this writing, it seems that little development has been accomplished since 2006 [6]. Zeitline is not presented as an alternative technology to the TimeKeeper methodology investigated as a part of this research, but as an example of an upstream consumer for the data TimeKeeper produces.

2.2.4 Other Forensic Tools

Building a timeline of events through the use of data available from the file system is a technique that has been examined with many open source and commercial tools developed for this purpose. Among the most well known are the toolkits mentioned in section 1.1.3: The Coroners Toolkit (TCT) [22], The Sleuth Kit (TSK) [14], EnCase by Guidance Software, and Forensic Toolkit (FTK) by AccessData .

TCT, developed by Farmer and Venema, uses the `mactime` tool, which in turn uses the `lstat()` system call to access the metadata times stored in inodes. TSK was written

by Carrier and is based on TCT [14]. It uses the `mac-robber` tool to retrieve the same information from files. Details about similar features in the commercial products EnCase and FTK can be found at [26] and [1] respectively. In [23], some UNIX commands are discussed that allow access to file metadata such as `ls -i`, `debugfs`, and the `stat` command.

Each of the previously mentioned toolkits has limitations and certain anti-forensics techniques have been developed to thwart evidence recovery. In an effort to cover their tracks, an attacker may use tools such as TimeStomp [35] or the Defilers Toolkit [46]. It has also been observed that the `touch` command can be used in an effort to cause confusion [23] [24].

In [19], a comparison is given between what the author considers to be traditional and advanced UNIX file systems. One of the main takeaways of this work is that the forensic model for the two classifications of systems, although related, is quite different when implemented due to the structure of the file systems.

In [23], the journal is mentioned as a source of time information and examples are shown of the information that can be extracted. TSK includes the `jls` and `jcat` tools which allow an investigator to obtain a listing of what is in the journal and a dump of a journal blocks respectively. TimeKeeper differs from these tools in that it procures information from the journal regularly, while the system in question is active, for forensic analysis after a security incident.

Due to the nature of how Ext3 zeros inode information, section 2.1.1, data carving cannot be ignored. In [13], data carving is defined as a process where a chunk of data is searched for signatures that correspond to known file types. This technique is necessary when investigating environments such as media that may not have file systems and unallocated portions of disks. There are a variety of techniques and tools that can be used to perform data carving, one of which is `foremost` [54] which does the operation in an automated fashion. `Scalpel` [41], another open source file carving tool, aims to maximize file carving performance through the use of various techniques. Although it is based upon a version of foremost, scalpel has proven to be much more efficent. It is not uncommon for a collection of the previously mentioned open source tools to be packaged together on a bootable CD

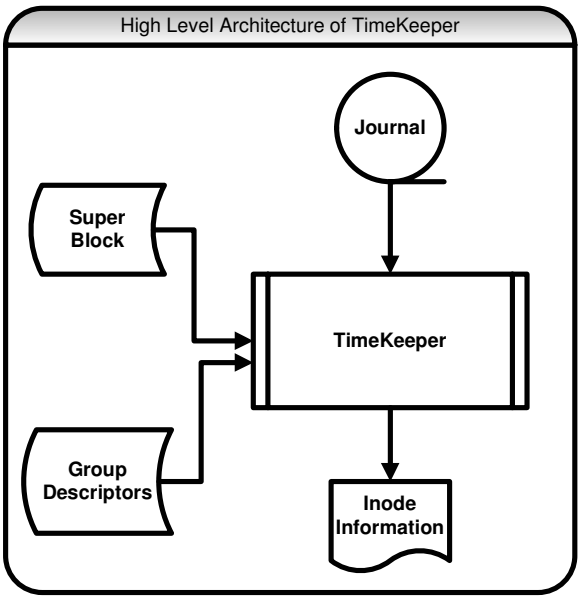

Figure 5: TimeKeeper Architecture

to create a forensic distribution such as the once open source Helix live CD [18].

2.3 TimeKeeper System Architecture

A major premise of this thesis is that because honeypots are different from production systems, additional sources of information can be explored which will aid in the forensic analysis of honeypots. File modification, access, and change (MAC) times are an example of this information. The theory tested in this chapter is that timestamp information can be made more reliable in investigations by devising new methods to log this data. In particular, the TimeKeeper method hypothesizes that in a honeypot environment metadata information can be extracted from the journal and stored for use in forensic analysis.

To test this hypothesis, a prototype has been developed based on the Ext3 file system that is able to successfully identify and extract inode structures from journal data. This is done by gathering the necessary data about the file system from the super block and group descriptors, shown in Figure 5, and then continuously reading and parsing through the journal data. From the inode structures, the metadata times are retrieved and exported giving the honeypot administrator a history of timestamps that can be used do to the following:

- Build a profile of what files are regularly touched
- Determine any irregularities
- Compare the inode times lifted from the system in question by one of the previously mentioned toolkits with those in the database to detect any nefarious changes that did not produce journaling entries

2.4 Methodology

The TimeKeeper system, for rapid development purposes, has been prototyped using the Python scripting language and makes use of a SQLite3 database for storage purposes. The details of the system are displayed in Figure 6. It runs in user space as an application and makes use of several well known file system tools. When initiated, the program first checks to see if the calling user has root level privileges. If not, this is considered an error, a message is displayed and the program exits. As displayed in Figure 6 beginning with the "Check User" block, this operational model inherently assumes a certain amount of trust in the administrative user. This trust may not always be present and a solution for this situation is proposed in Chapter 3.

Command line options were necessary in order to increase usability across different experiments. The arguments are parsed with the Python getopt.getopt() function. Currently, the following options are available:

- -m, –memory

  This sets a flag stating whether the database should initially be built in memory or flushed to disk from the beginning of the program. The first version of the program setup the database and flushed to disk every time the journal was checked. This presented a problem when the mechanism was combined with system call introspection in [56]. During a flush to disk, the system call introspection output would be overwhelmed with the data caused by TimeKeeper and effectively miss all other system data. By keeping the database in memory until a prescribed time or number of polls, the problem is avoided.

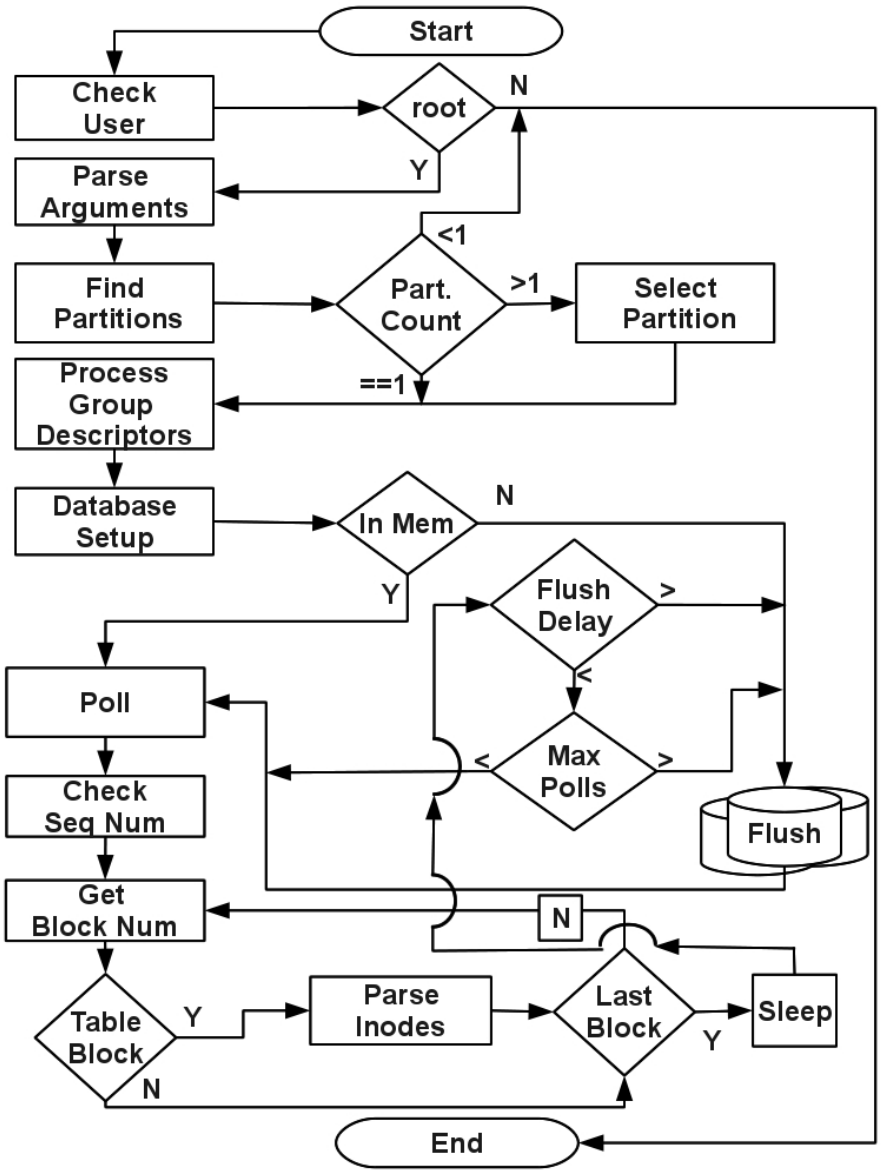

Figure 6: TimeKeeper Flow Chart

- -o [filename], –output [filename]

  This is the name of the database that will appear on disk. If this is not set, the default behavior is to name the database "TimeKeeper". If the memory flag is set, the default behavior is to name the database "FromMemDB".

- -P [number], –polling_interval [number]

  This option sets the number of seconds to wait between checking the journal.

- -p [number], –polls [number]

  This option sets the total number of times for the journal to be checked before exiting the program. If not set, TimeKeeper will by default run in an infinite loop.

- -d [number], –delay [number]

  This is the maximum number of seconds to wait before dumping to harddisk. This option is necessary when the memory flag is set. If not provided, the in-memory database will be flushed to disk after every poll. This makes the default behavior equivalent to simply creating an on-disk database.

In Figure 6, the "Find Partitions" block makes use of the `/etc/mtab` file to determine if any Ext3 partitions are mounted on the system. If more than one is present, the user is prompted to select which partition to track. If the there is only one partition, it is selected by default and the program continues. In the event there are no Ext3 formated partitions, a message is displayed to the user and the application exits.

All of the preceding can be considered information necessary to identify and describe the environment and mode of operation in which the monitoring is occurring. The next sequence of steps form the core of the system. First, described by the "Process Group Descriptors" block in Figure 6, the file system super block and group descriptors are parsed. This information is obtained through the use of the `dumpe2fs` command. Examples of important information in the file system super block include the block size, the number of blocks per group, the number of inodes per group and the number of inode blocks per group. This information is necessary to calculate inode numbers as the numbers are not stored anywhere on the disk. Another thing to note is that the super block also contains the inode associated with the journal and the size of the journal. Each group descriptor is examined to find the offset into each block group, in terms of blocks, at which the inode table begins and ends. This offset value varies as only a fraction of the block groups will contain backups of the super block and group descriptors. This step only needs to happen once during a logging session. For that matter, if the same partition is being monitored across multiple logging sessions and no changes to the underlying file system structure have been made, then this

information can be saved to the database and this step need not occur again.

The next major step is the creation of a SQLite3 database for information storage and retrieval. It is at this point that the memory flag is checked and influences the flow of the overall program. Either way database tables and indexes are created, but exactly when the information gets written to disk is different. By default, the database gets dumped to disk immediately, denoted by the "Flush" operation in Figure 6, and this file is updated each time the journal is checked. If the database resides in memory, two counters are used to determine whether it will be dumped to disk. The first counter uses the delay input argument and the second uses the polls argument. Once the delay has been exceeded, the memory database is flushed to the disk, the program then sleeps, and polls again reseting the variables tracking the time since the last write. Once the user specified number of polls has been exceeded, the database is dumped to disk and the program exits. These counter operations are represented in Figure 6 by the "Flush Delay" and "Max Polls" decision blocks respectively. The reason for this setup is that during an experiment, a person may not want to write to disk until the completion of certain operations. In this instance, a very large delay can be set and a maximum number of polls can be set so that the flush occurs after these events. However, this mode of operation risks all historical data being lost if for some reason the TimeKeeper application is stopped beforehand or the complete computer system is reset.

After the database tables have been created, the "Poll" subroutine block in Figure 6 is entered. In this process, the contents of the journal are acquired via the `debugfs logdump` command. The output of `logdump` is parsed so that transaction sequence numbers are obtained. These are not only useful in helping to identify the MAC times of a particular inode at a moment in time, they also aid in processing the journal data. When the initial dump happens, there is no information in the database so all of the output is parsed and stored. However, after the initial dump the last sequence number that has been extracted is stored in a variable. Unless the sequence restarts or rolls over, all transactions lower than this sequence number have already been stored and can be ignored until fresh data is found. Although this avoids the problem of reprocessing data that has already been gathered, it

does not solve the problem of the journal wrapping around before the next polling cycle.

Next, in the Figure 6 "Get Block Num" block, the file system block number is parsed from the output. The number is checked against the group descriptors to determine whether the block is in the inode table. If not, the rest of the output is skipped until the next file system block is encountered. Once a block that is in the inode table has been located, the "Parse Inodes" process block is entered and the first inode number is calculated via the following formulas:

$$GroupNum = \frac{BlockNum}{BlocksPerGroup}$$

$$BlockOffset = BlockNum - Group[InodeTableStartBlock]$$

$$InodeNum = (InodesPerGroup * GroupNum) + \frac{BlockSize}{InodeSizeBytes} * BlockOffset + 1$$

This initial number is incremented as each inode in the block is parsed. The advantage of capturing information from the journal is that the journal stores information at block resolution before it is transfered to the file system area. This means that if one inode in a block gets updated, all of the other inodes in that block will get logged as well. This also serves as a disadvantage as there is no rapid way to identify which particular inodes in a block have been updated. One method to circumvent this problem is to query the database for each inode that is discovered and compare the latest results returned with what has been captured before inserting the information into the database. The method used by the TimeKeeper protoype is to attempt a database insertion for every inode and discard duplicate entry errors.

This process is repeated until the last block in the journal has been checked. After the journal has been logged, the flush delay value is checked against the current time. If it is not time to flush and the number of polls left is greater than zero, the program sleeps for a time specified by user input before polling the journal again.

2.5 Testing

2.5.1 Experiment Setup

To test the TimeKeeper prototype, a loopback device was created and mounted as an Ext3 partition inside of a Ubuntu Linux virtual machine. TimeKeeper was executed outside of

Table 1: Sequence of Events

Event	Command	Sequence
1	touch	21
2	edit (VIM)	23-26
3	touch	27
4	echo	28
5	mv	30
6	rm	31

this partition at the interval of 5 seconds between journal dumps, while various types of file modifications were performed and tracked. It is very important that the database be located outside of the file system that is being monitored to avoid a feedback loop.

2.5.2 Results

As a first test to show that historical modification, access, and change times could be obtained, a file was created with the `touch` command. This created an empty file with the modification, access, and inode change times set equal to each other. This was verified by the `stat` command and may be seen in sequence number 21 in Table 2. This file was then opened and edited by the VIM text editor. The text editor created a copy of the original file that was then edited and saved as a different inode. When a file is saved in VIM, the original file is deleted and the new inode assigned the filename. Table 1 shows the sequence numbers that map to these events.

The `stat` command, after the editing of the file by VIM, showed that the filename is associated with a new inode that had a new set of metadata times. When the TimeKeeper database was queried based on the original inode number, it clearly showed the life span of the first inode including its deletion time. A similar query based on the new inode number showed a series of events that ends with all of the metadata times set to the same value making it appear to be a new file. When these results are interspersed, a pattern between the two inodes emerges and it becomes apparent that the deletion time of the first inode corresponds with the change time of the second inode, yielding a more accurate picture of the file history. This is shown in Table 2.

The file was then accessed again using the `touch` command which modified all of the

Table 2: TimeKeeper: `touch` and `vim` output

Inode	Seq	Mtime	Atime	Ctime	Dtime
12	21	t1	t1	t1	-
14	21	-	-	-	-
12	23	t1	t2	t1	-
14	23	t2	t2	t2	t2
12	25	t1	t2	t3	-
14	25	t3	t3	t3	-
12	26	t1	t2	t3	t3
14	26	t3	t3	t3	-

metadata times by setting them to the same value. TimeKeeper also recorded this modification.

Next, the word Hello was appended to the end of the file using the `echo` command. This caused the inode modification and change times to be updated while the access time remained the same. This happened because the file was never actually opened; instead the bytes were added to the end of the file as denoted by the information stored inside of the inode.

When the file was renamed using the `mv` command, the inode number remained unaltered as well as the access and modification times, but the change time was altered. Finally the `rm` command was used to delete the file, thereby causing an update to the deletion time as well as the modification and change times. All of these incremental file changes were detected and captured by TimeKeeper and are summarized in Table 3.

2.5.3 Conclusions

The results of section 2.5.2 show that the TimeKeeper method for MAC time preservation is valid. The metadata timestamps were successfully harvested from the journal during the experiment and provided a history of the events which transpired. Without this history,

Table 3: TimeKeeper: `touch`, `echo`, `mv`, `rm` output

Inode	Seq	Mtime	Atime	Ctime	Dtime
14	27	t4	t4	t4	-
14	28	t5	t4	t5	-
14	30	t5	t4	t6	-
14	31	t7	t4	t7	t7

the relationship between inodes 12 and 14 may not be readily apparent if an investigator were only given the last two entries of Table 2. This is often the case when a hard disk image is examined. Also, because all changes in metadata time recorded to the journal are captured with this technique, usage of the `touch` command to reset MAC timestamps was detected as shown in first line of Table 3. This counters the usage of the `touch` command, and other similar tools, in anti-forensic techniques that seek to obfuscate an adversary's actions through timestamp manipulation.

2.6 Limitations

The current prototype is written in the Python scripting language. Although this provides a means for rapid development for a proof of concept, it does have certain drawbacks. For this technique to gather information, the program must constantly run in the background. If an attacker is skilled enough and truly wants to hide any actions taken, the program can be detected and stopped. For these reasons, a kernel patch or module might yield better results as they operate closer to the file system. Some contemporary research along this line of thought is breifly covered in Section 3.2.4. While the general technique covered in that section uses modified system calls, a low level TimeKeeper implementation would instead modify the journal block device.

Since this technique depends on the presence of the journal, it is also limited by the information stored therein. This means that the data that can be obtained is directly influenced by the mode of journaling supported. If an attacker wants to avoid any journal probing device, then they could circumvent the file system writing directly to the raw device. This technique, as mentioned in [23], may cause file system corruption if used incorrectly which may be a tell-tale sign of suspicious activity.

Finally, the current work is limited to the Ext3 file system. This issue is not paramount as the theory behind the technique can be extended to any journaling system and Ext3 is pervasive in the Linux community. Even with the fairly recent adoption of Ext4, Ext3 file systems should remain in use for some time.

2.7 Futurework

Currently the proof of concept prototype is set to poll the journal at a predetermined interval. The rate at which the journal overwrites itself is dependent upon the type of journaling supported by the file system, the type of operations being performed, and the rate at which file system operations are performed. This approach is not optimal as the rate increases. A better solution would be to monitor a certain block in the journal such that every time that the block is accessed for writing, the journal is automatically audited by the monitoring program. This tactic would prevent missing journal data due to excessive file operations between sampling times. Also, it would decrease the overhead required in finding the synchronization point between the current and previous journal samples.

Since the data being gathered can become untrusted if it is stored on the same medium that the attacker has access to, methods must be implemented to efficiently to protect this history. Some of the proposed solutions include storing the data in hidden partitions and writing to write once media such as CD-Rs and DVD-Rs. Furthermore, although this method of gathering more insight leverages the honeypot characteristic of being free of production constraints, the exact performance and storage penalties have not been calculated. Although the idea proposed could be implemented on production systems, the benefits gained would not justify the cost incurred. By obtaining performance measurements in a honeypot, optimizations might be made so that similar systems could be developed to aide in real-world situations where the environment cannot be controlled.

This method for building a metadata time history for a richer source of forensic information is not intended to be a stand alone solution. It is a component and first step toward building a forensic framework that specifically suits the honeypot environment which can be further extended as these environments continue to evolve. In [55] a method is proposed for the monitoring of system calls to determine whether an encrypted application can be trusted. Combining the techniques discussed there with TimeKeeper could yield a system that produces greater forensic evidence. This line of research was first explored in [56] and can be continued. Furthermore, as Ext4, discussed in section 2.1.3, has made changes to several on-disk structures in the file system as well as in the journal area, work should be

done so that the TimeKeeper system will be compatible with the new file system.

CHAPTER III

DENTRY LOGGING

Honeypots are currently deployed in a variety of networks as reconnaissance devices. Until recently, Ext3 was the default file system for most Linux distributions; therefore, it is only natural that it be fully examined as a potential environment. However, due to its design Ext3 can inhibit an investigation under certain circumstances. In this chapter, the design of a method for honeypots is presented that takes advantage of the Virtual File System Layer in Linux to provide an additional source of information in forensic frameworks. This new technique allows the translation of inode numbers to filenames in a historical context thereby providing a forensic analyst with a better picture of what has transpired. This work proposes monitoring low-level operating system functions as opposed to considering information from the file system alone.

The mapping of filenames to inode numbers in the Ext3 file system is a unidirectional operation consisting of locating a filename in a directory and determining the associated inode number. This commonplace procedure is more than adequate when considering perfunctory user activities such as file creation, editing, saving, and deletion; however, certain specialized usage cases benefit from a reverse translation being performed. The structure of Ext3 inhibits the retrieval of a single or set of filenames given only an inode number. The `findutils` and `e2fsprogs` suites along with system call monitoring [55] can be employed to accomplish this task, but these utilities can only provide an inode to filename translation at the time they are executed. This limited scope may obscure details which are pertinent in a investigation. In this chapter, `findutils`, `e2fsprogs`, and system call monitoring are examined to determine how they have historically been used to retrieve a filename given only an inode number. The knowledge gained was used to develop a method for determining a filename of a given inode. This increases the semantic meaning of the data produced by a system that captures inode information alone.

The proposed method takes advantage of the Virtual File System layer in Linux in order to gain access to memory structures known as dentries. By exploiting the dentry caching system, information can be logged in real-time. This data can then be correlated with metadata that has been archived. This approach calls for insertion of code into the Virtual File System layer which increases the difficulty of disabling any logging mechanisms in the event that the target system is compromised.

The inode mapping system described in this chapter is designed specifically with honeypot environments in mind. In these environments, attackers may escalate their privilege level so that even administrators cannot be trusted. This requires the design of a logging mechanism that cannot be easily circumvented or deactivated by the root user. When these considerations are combined with the fact that honeypots are known for running customized logging software and kernels, the argument for kernel modification is strengthened.

The remainder of the chapter is arranged as follows. Section 3.1 contains a brief overview of the Linux Virtual Files System and the TimeKeeper metadata archival system is summerized in section 3.1.2. In section 3.2, related work such as system call monitoring is discussed. Sections 3.3 through 3.7 detail the architecture of the proposed method, display the procedure, and examine results of preliminary experiments respectively. Future directions for this research are presented in section 3.8 before finally concluding in section 3.9.

3.1 The Virtual File System

As this chapter discusses a method of performing inode number to filename mappings for forensic purposes, a basic understanding of the underlying technology is a necessity. The following provides a brief discussion of the Virtual File System layer with the intent of clearly presenting the problem faced. For the purposes of showing how the general technique described could be used, the TimeKeeper architecture is briefly described as a consumer of the inode to filename translation system. In order to fully appreciate the problem, knowledge of the Extended File systems is necessary and was presented in section 2.1.1.

In Linux, a parallel can be drawn between the Virtual File System (VFS), first introduced to the UNIX world via SunOS2 in 1985 [29], and the Internet Protocol (IP). Like IP, the VFS permits many processes to work above it by providing a common interface through which they can perform I/O operations while hiding the details of the underlying file system that is actually resident on the medium. This gives Linux flexibility in that it can support any file system that is VFS compatible without having to make changes to applications. So, like IP, a great amount of development can happen above and below the VFS layer which makes changing the layer itself somewhat risky [31][5].

3.1.1 Dentries

The VFS is heavily UNIX influenced and thereby supports these UNIX-like file system structures: the superblock, the inode, the dentry, and the file. Of these structures, the dentry is unique in that it is not derived from an on-disk structure. It is created by the VFS while performing directory operations. A dentry is a directory entry and not a directory itself. A directory is simply viewed as a file. In the path /home/user/test.txt, the components /, home, and user represent directory files while test.txt represents a regular file. All of these objects have inodes which represent them on disk. When this path is being resolved, a dentry is created for each component of the path.

The operations that can be performed on a dentry are defined by the dentry_operations structure which is included in the dentry structure as the d_op field in the dentry. The dentry_operations structure contains a set of function pointers that can either be redefined by the underlying file system implementation or left to perform the default VFS action.

Dentries have three states: used, unused and negative. A used dentry means that the object has a valid inode and that it is in use. When unused, the dentry still has a valid inode, but it is not being used. A negative dentry does not have a valid inode. This can be the case if the inode is deleted or if the path provided for a lookup was incorrect. These states control whether a dentry is retained or not. Used dentries cannot be discarded, while unused dentries are only discarded in the case memory is needed. Negative dentries are the first to be discarded when space is needed as they possess the least amount of priority

because they are invalid anyway. Dentries are stored in the Dentry Cache (dcache) to decrease the time it takes to resolve a path.

The dcache structure is comprised of lists of used dentries, least recently used dentries, and a hash table. The lists of used dentries are connected to the VFS inode object's i_dentry field. A list is necessary to account for the fact that an inode can have several links pointing to it resulting in it having multiple dentries. The least recently used list is comprised of valid unused and negative dentries that are reverse time sorted so that the oldest dentries are toward the tail end and can be reclaimed as necessary. The hash table is an array of pointers to lists which contain dentries that have equal hash values [31]. This structure allows the VFS to quickly determine whether a dentry is cached by using its hashed value. The dcache is important for the following reasons:

- It reduces the amount of time necessary to resolve frequently used paths by eliminating the need for the VFS to manually resolve each path component starting at the root directory entry (file system walking).
- The dcache helps to control the inode cache as, according to [31], the dentry maintains a positive usage count over the inode. This means that as long as a dentry object remains in the cache, the corresponding inode will also remain in memory in the form of the inode cache. This eliminates the time necessary for the VFS to allocate and instantiate that inode.

More information about the dentry cache can be found in [5].

3.1.2 TimeKeeper

TimeKeeper is an example of an application that can benefit immensely from the ability to map inode numbers to filenames throughout a time period. Presented in [21], the metadata archival system is designed for the purpose of addressing anti-forensic efforts on honeypots by preserving file metadata such as the modification, access, and change (MAC) times in the Ext3 file system. The anti-forensics efforts described in [24] include specific attacks on metadata time information using known tools such as TimeStomp [35] which specifically

targets timestamp information in the NTFS file system. As shown in Figure 3, the TimeKeeper technique used to gather metadata takes advantage of the Ext3 journal. The super block and group descriptor information is acquired as a means to identify a particular block that is seen in the journal and determine what actions are to be taken. This method is not intended to be a stand alone solution, but a piece in a larger forensic framework that has the goal of providing a richer set of evidence when performing the analysis of a honeypot.

One of the limitations of this archival method is that it currently does not gather filenames when extracting inode information from the journal. This lack of functionality is mainly due to the fact that in Ext3 the filenames are stored in directory structures. Directory data is actually stored along with file data. Thus the only way to retrieve this information is to first process an inode, detect its file type, and then gather a list of directory file system blocks. Otherwise, every file system block seen by the journal will have to be processed. Also, if an attacker is skilled enough, he or she may be able to circumvent the generation of journal traffic by writing directly to the device. However, this may produce inconsistent timestamp information as well as possibly damage the file system on the disk. Attackers could also resize or remove journaling from non-root volumes to limit the amount of information gathered. In this chapter, TimeKeeper output will be analyzed in conjunction with logged dentry data to demonstrate the added value that VFS logging provides.

3.2 Related Work

3.2.1 find and debugfs

As noted in [44] the `find` command, part of the `findutils` package, and the `debugfs` utility, included in `e2fsprogs`, have the ability to translate an inode number to one or more filenames. Through the use of the -inum option, the `find` command is able to walk down a directory structure beginning at a location specified by the user. If it is assumed that the user does not have root privileges, then certain areas of the file system will be off limits to them. The `debugfs` environment, which is used to troubleshoot file system problems such as corruption, can perform the same action through use of the `ncheck` command.

It has been noticed that when attempting to do an inode to filename translation with these utilities, the first operation can take a longer amount of time than subsequent mappings to the same inode number or closely located inode numbers. The observed speed increase in the secondary lookups is most likely caused by the dcache having had entries added during the initial search. If the inode data provided by a user or application needing to perform a translation is ordered in such a way that queries are executed according the layout of the directory structures, then the use of `find` or `debugfs` may present a valid method to perform lookups. However, the tools only provide insight into the state of the file system at the instant a command is executed. To overcome this limitation, a translation attempt would need to be made every time an unknown inode number is seen. The inode numbers produced in a short period of time may not belong to the same branch of the overall directory tree. For example, a file on a user's desktop may be opened with an application located in the /usr/bin directory which may in turn depend on a number of libraries. Even if physical locality is not exploited to save time when performing translations, the situation is exacerbated by the effects that that real-time searches would have on the dentry cache.

Multiple searches for disparate inode numbers can cause valid dentries to be retired from the cache early. This dentry churn will have the effect of slowing down the system as full pathname resolutions must occur more often. Overall the `find` and `debugfs` commands do not adequately meet the design requirements of providing inode number to filename translations with historical context.

3.2.2 Virtual Machine Monitor Sensors

In [2], a technique is presented where virtual honeypots are monitored using sensors. The sensors have the ability to transfer control from virtual kernel to the virtual machine monitor where the data provided by the calling function can be used for further analysis thereby classifying it as a type of virtual machine introspection. Of particular interest is the inode access sensor which takes advantage of the VFS. The sensor obtains the directory entry of a file that is being opened, which is used to obtain the absolute path of the file. The purpose behind this design is to detect malicious activity affecting sensitive files. If an

attacker wants to determine whether or not they are in a virtualized environment, methods presented in [28] may be employed for detection. The approach taken in [2] verifies that the VFS is a prime target to be exploited for file information.

3.2.3 System Call Modification

System call monitoring, replacement, or modification is a technique that has been used in many applications. Sebek [48], for example, can be used on honeypots to capture encrypted session keys and keystrokes. It works by modifying the behavior of the sys_read function. In [55], many different system calls are modified to monitor the behavior of encrypted programs. This approach can have shortcomings as it has been noted in [25] and [2], that system call monitoring can have potential path resolution problems. For example, it has been observed that at the system call level one might see open("./config", O_RDWR). The difficulty that arises here is that there is uncertainty as to where this file truly is located. By logging dentry information, uncertainty is overcome due to the fact that the parent dentry can easily be identified. Thus starting at the file config, one can obtain a fully canonicalized path by recursivly identifying parent dentries.

3.2.4 MAC Trail

The MAC Trail system [3] has a similar purpose to the previously mentioned TimeKeeper system. It provides an enhancement to Ext2 in the form of a loadable kernel module to preserve MAC times. The approach used implements a mechanism at the VFS layer which augments the behavior of the sys_open system call. Furthermore, the data collected is hidden from system users through utilization of custom data structures that store the information to the disk in random blocks.

One strength to this approach is that it can implemented on Ext2 and Ext3. Also, by monitoring system call information, the filename and inode number may be retrieved simultaneously. However, the model used assumes that the system administrator can be trusted. This is not a safe assumption with honeypots. In addition, as it uses a form of system call monitoring, it can fall prey to path resolution complications.

This technique is improved upon in [15]. Here, the authors are only concerned with

monitoring a specific set of files and again choose to take advantage of the sys_open call. They obtain further information by monitoring the unlink system call. While allowing their system to detect file deletion, this move also prevents the deletion of the log file. Instead of taking the approach used in [3] to store the log file, they instead monitor the getdents() system call to hide the presence of the log file. Clearly, this assumes a certain amount of trust for the root user. However, if the admin account is compromised, the log file cannot be easily found as the `ls` command depends upon the getdents() system call. As with the MAC Trail system, this approach uses a loadable kernel module. Thus if module is detected and unloaded, the technique becomes ineffective. This work further reinforces the benefits of capturing data at the VFS layer.

3.3 Theory/Architecture

The translation of a filename to an inode number takes place at the VFS layer, making this the primary target for gathering data to perform the reverse operation. When a path is being resolved, dentries are created for each of its components. These dentries are held in memory to form the dcache which increases the speed of frequently used searches. The operations that can be performed on a dentry object are defined in the d_op field of the structure, which points to the dentry_operations structure, as well as in the fs/dcache.c file of the kernel source. The location of both dentry and dentry_operations structures is in the include/linux/dcache.h file.

The function pointers defined inside of the dentry_operations structure can be redefined by the implementation of a particular filesystem, which allows that file system to handle any peculiarities the default VFS operations do not address. It is therefore necessary to add a logging mechanism to the functions defined outside of this structure to make changes more robust across differing file systems. As of June 2007, the dcache API consists the following major functions:

- dget - increments the usage count of a dentry
- dput - decrements the usage count of a dentry

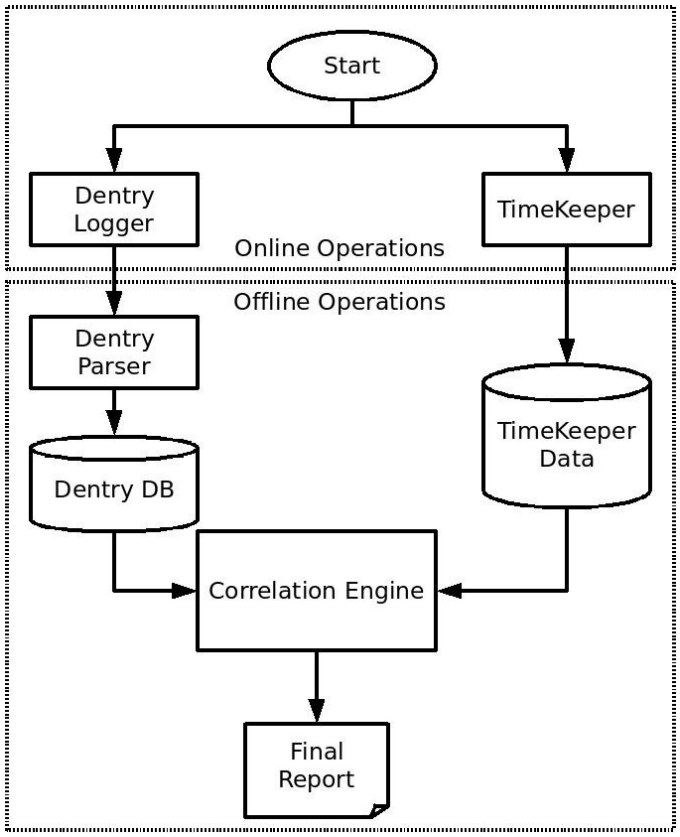

Figure 7: Testing Architecture

- d_drop - removes dentry from the parent hash list
- d_delete - deletes a dentry
- d_add - adds a dentry to the parent hash list
- d_instantiate - adds dentry to the alias hash list and updates inode information
- d_lookup - finds a dentry given the parent and path name

By adding logging mechanisms to one or more of these functions, a more detailed picture of the events recorded in the Ext3 journal can be obtained. The basic operation of the mapping system is displayed in Figure 7. This figure denotes the actions that take place while the system in question is active versus the processing which should be handled offline. The journal metadata archival system and dentry logger are executed in tandem so that the journal data can serve as a starting point to identify a window of interest in the VFS

data, which is much more extensive in comparison. Simultaneous execution also serves to check the validity of the data received by the dentry logger.

3.4 Prototype

The dentry logging mechanism was prototyped through modification of the Virtual File System kernel code. In particular, it was the d_instantiate function which was altered for experimentation because, generally, this is where the inode information is filled into a dentry structure. The collected data was sent to the kernel message buffer and written to a log file. The logs were parsed offline using python scripts and imported into SQLite3 databases. This approach provides the following benefits over the use of a kernel module:

- As the changes have been made to the actual kernel code and compiled, there is no need to take steps to hide a kernel module, which is the case in [48] and [15].
- By making changes to kernel code rather than implementing a kernel module, the hidden module detection techniques discussed in [17] can be avoided.
- The use of a kernel module, as presented in [3], assumes that the administrator of a system can be trusted. This assumption cannot be made when considering systems such as honeypots. As a reconnaissance mechanism, the honeypot only gains value after it has been compromised so that the exploits used to gain access can be studied. Furthermore, over the course of an attack, an intruder may escalate their privilege level. Therefore, an administrator cannot be trusted with the ability to disable this logging mechanism.

The experiments made usage of the Linux 2.6.23 kernel. Based on its source code, a library of functions with the primary one being the dentry_print function was developed. Dentry_print() takes in as arguments a dentry object, an inode object, and a string which denotes the name of the function from which is called. This is the only function call which is inserted into the dcache.c file. The dentry_print function then attempts to access the filename associated with the dentry object before validating the the inode object and

calling the print_inode function. As each dentry object contains a pointer to its parent dentry, this data structure is also accessed and recorded.

The print inode function accesses the VFS inode object and gathers the inode number, inode mode, as well as the timestamp values. These timestamp values differ from the values in the Ext3 inode object in that the VFS inode object does not have a dtime field. However this drawback is balanced by the fact that VFS inodes contain time values that are recorded in the subsecond range unlike their Ext3 counterparts. Once the information from the dentry and the parent dentry have been recorded the control is returned to the calling function. This form of implementation allows minimal changes to the actual kernel code and while providing a method to quickly insert the logging function into multiple places to determine the best vantage point given a set of objectives.

Due to the nature of the honeypot environment, the root account cannot be trusted. This creates a need to export the logged information from the honeypot in question. Two approaches have been examined in this effort. The first involves writing to a device from within the kernel while the second uses a remote logging server.

In the prototype implementation, the dentry_print() and print_inode() functions have the ability to export information via serial port. This allows information, a sample of which is shown in Figure 8, to be gathered during the boot process and sent to a remote machine. This method was successfully tested using a QEMU virtual machine which output the data to a file. However, when tests where run using two real machines, a noticeable amount of delay was introduced. Although exportation by means of the serial port proved not to be a viable option due to the data rate limitation, the data gathered justifies the concept of writing directly to a remote device. Going forward, data exportation through higher speed ports and other techniques should be investigated.

In [38], it is recommended that a remote logging server be setup as one of the the more important steps of pre-incident preparation. As honeypots need to mimic production systems, this procedure is valid as a means of information preservation. By using printk() statements in the dentry_print() and print_inode() functions, data can be logged to the kernel message buffer. This data will then be processed and appear in the system log files

```
Caller: d_instantiate
    Inode Number: 1
    Inode Mode: 41ED
    1232681240l 84005250l (mtime)
    1232681240l 84005250l (atime)
    1232681240l 84005250l (ctime)
    Filename: / , /
Caller: d_instantiate
    Inode Number: 1
    Inode Mode: 41ED
    1232681240l 88005500l (mtime)
    1232681240l 88005500l (atime)
    1232681240l 88005500l (ctime)
    Filename: / , /
Caller: d_instantiate
    Inode Number: 1
    Inode Mode: 4180
    1232681240l 88005500l (mtime)
    1232681240l 88005500l (atime)
    1232681240l 88005500l (ctime)
    Filename: bdev: , bdev:
Caller: d_instantiate
    Inode Number: 1
    Inode Mode: 416D
    1232681240l 92005750l (mtime)
    1232681240l 92005750l (atime)
    1232681240l 92005750l (ctime)
    Filename: / , /
```

Figure 8: Serial Device Output

located in the /var/log/ directory. Although the syslog daemon that is installed by default with Ubuntu 8.04 allows the configuration of remote logging, it currently only supports transfer via the UDP protocol. As the primary purpose of a honeypot is to gather information, more reliability is needed in order to prevent data loss. The solution chosen for this problem was to install the rsyslog package on the honeypot and logging server. Rsyslog provides many features such as database support and transportation via SSL, but for these experiments only TCP was used. It should be noted that other software such as syslog-ng could also be used for this functionality.

The logging server approach has the benefit of being easier to implement when compared to investigating and implementing low-level data exportation via device drivers. Also, as

```
Mar 16 14:52:42 Random mysqld_safe[11209]: ended
Mar 16 14:52:45 Random kernel: Kernel logging (proc) stopped.
Mar 16 14:52:45 Random kernel: Kernel log daemon terminating.
Mar 16 14:52:47 Random rsyslogd: [origin software="rsyslogd"
swVersion="1.19.12" x-pid="4820"] exiting on signal 15.
Mar 16 14:53:33 Random rsyslogd: [origin software="rsyslogd"
swVersion="1.19.12" x-pid="4316"][x-configInfo udpReception="No"
udpPort="514" tcpReception="No" tcpPort="0"]
restart
Mar 16 14:53:33 Random kernel: rklogd 1.19.12, log source = /proc/kmsg started.
Mar 16 14:53:33 Random kernel: nsec
Mar 16 14:53:33 Random kernel: [36.887286] ctime: 1237229613 sec 345744674 nsec
Mar 16 14:53:33 Random kernel: [36.887292] Caller: d_instantiate
Mar 16 14:53:33 Random kernel: [36.887294] Inode Number:13314
Mar 16 14:53:33 Random kernel: [36.887295] Inode Mode: 040755 0x41ED
Mar 16 14:53:33 Random kernel: [36.887297] File Type: Directory
Mar 16 14:53:33 Random kernel: [36.887299] mtime: 1237229613 sec 349746390 nsec
Mar 16 14:53:33 Random kernel: [36.887301] atime: 1237229613 sec 349746390 nsec
Mar 16 14:53:33 Random kernel: [36.887303] ctime: 1237229613 sec 349746390 nsec
Mar 16 14:53:33 Random kernel: [36.887305] Filename: queue, queue
Mar 16 14:53:33 Random kernel: [36.887307] Parent Inode:
Mar 16 14:53:33 Random kernel: [36.887309] Parent Filename: .udev, .udev
Mar 16 14:53:33 Random kernel: [36.887312] Parent Inode Number:1141
Mar 16 14:53:33 Random kernel: [36.887314] Parent Inode Mode: 040755 0x41ED
Mar 16 14:53:33 Random kernel: [36.887315] Parent File Type: Directory
Mar 16 14:53:33 Random kernel: [36.887317] Parent mtime: 1237229613 sec
349746390 nsec
Mar 16 14:53:33 Random kernel: [36.887319] Parent atime: 1237229612 sec
225652715 nsec
Mar 16 14:53:33 Random kernel: [36.887321] Parent ctime: 1237229613 sec
349746390 nsec
```

Figure 9: Rsyslogd Messages

the honeypot and the logging server have close physical proximity to each other, a local area network can be established to reduce the delay caused by competing network traffic. These benefits are balanced by the fact that until the rsyslog daemon is started, no traffic will be sent from the honeypot to the logging server. During this time period the kernel message buffer can easily overwrite itself several times resulting in data loss. This can be seen in Figure 9 which displays the messages in the /var/log/syslog file. In line 10 of Figure 9, the log file displays nsec as a message from the kernel. In line 6 of the log file, the entry shows the rsyslog daemon restarting. To ease the task of parsing the log files and verify the data produced by the dentry logging functions, headers are added to each data field. The

first complete dentry record shown starts at line 12 with the heading Caller: and continues until the last line of the figure. Lines 10 and 11 are the remnants of a logged dentry that has been partially overwritten due the circular nature of the kernel message buffer.

As shown beginning with line 12, each complete dentry record contains the following fields:

- Caller

  Displays which function has been instrumented, thereby allowing the identification of the type of information that can be retrieved from a given function. This field also aids in calculating the frequency with which a function is called.

- Inode Number

  Shows the VFS inode number to which the dentry is attached.

- Inode Mode

  This is a numeric value corresponding to the file type of the VFS inode.

- File Type

  Holds the decoded value of the Inode Mode field stating in plain text the type of file with which the dentry is associated.

- mtime

  Shows the seconds and nanoseconds from the epoch that the file was last modified.

- atime

  Shows the seconds and nanoseconds from the epoch that the file was last accessed.

- ctime

  Shows the seconds and nanoseconds from the epoch that the file metadata was last changed.

- Filename

  Displays the filename associated with the inode number.

- Parent Inode

  This is a marker heading denoting the start of data retrieved from the parent of the current dentry. All of the Parent fields following this line have the same purpose of their non-Parent counterparts.

3.5 Experiment

In order to test the implementation of this technique, the following experiment consisting of several simple file operations was performed:

1. Experiment start time recorded
2. A test directory was made
3. A test file was created in the test directory with the `touch` command
4. The test file was edited using `vim`
5. A secondary test file was created from the first test file with the `cat` command
6. The inode numbers of the directory contents were recorded using `ls -ali`
7. The first test file was deleted using `rm`
8. The end time of the experiment was recorded

A five second pause was inserted between steps to ease the process of manual analysis.

3.6 Results

In Figure 10, the output of the experiment log file is displayed. The start and end times of the experiment appear at the top and bottom of the file respectively. The timestamps are

```
1237237019
229380 .
229377 ..
229383 Test1.txt
229381 Test2.txt
1237237064
```

Figure 10: Condensed Log File Contents

in the form of seconds since the UNIX epoch. This output is only a snapshot of the testing directory immediately preceding the final two steps of the experiment. Using this as a guide to start an analysis of the results, the data archived using the TimeKeeper technique was examined.

As TimeKeeper is an existing system that archives journal data, it is used to provide sample data, which can then be augmented using dentry logging. Since we are logging information at the VFS level, similar output to the archived journal data can be produced in a different manner. For example, by enduring a greater performance sacrifice through implementing more taps, such as in the d_hash function, a more complete picture of dentry searches can be obtained. This will also create more extensive logs than those produced by journal archiving. However, the scope in this research is inode to filename translation which only requires modification of the d_instantiate function. This imposes a limitation of not being able to track incremental dentry changes which would be possible by instrumenting other functions. If the Ext3 journal has not been archived, dentry logging will still provide the benefit of enabling inode to filename translation of files that have been deleted whose inode structures have yet to be recycled.

Figure 11 shows the abbreviated output of the archived journal data when it is imported into a database and queried for events during the specified time period. The data fields that are gathered from the journal but which are not shown here include the file system block number, the journal sequence number, and the journal block type. To further reduce the set of data returned by the query, the output is filtered for any inode in the 229000 to 229999 range. From left to right, the columns in the figure consist of the inode number, file modification time (mtime), file access time (atime), file change time (ctime), and file deletion time (dtime) recorded for an entry. The entries that correspond to the inodes listed in the log file are in bold type. At this point, the filenames corresponding to inodes 229378, 229379, and 229382 are completely unknown. Although there is some insight into the filenames of the bolded entries, it is not possible to say what filename maps to a particular inode number at a given instance in time using the data gathered from the journal alone. It will soon be shown that this is due to inode recycling.

inode Number|mtime|atime|ctime|dtime

229377|1237237019|1237237015|1237237019|0
229379|1237237019|1237237019|1237237019|0
229380|1237237019|1237237019|1237237019|0
229378|1237237007|1237237024|1237237007|0
229380|1237237024|1237237019|1237237024|0
229381|1237237024|1237237024|1237237024|0
229378|1237237007|1237237029|1237237007|0
229380|1237237030|1237237019|1237237030|0
229381|1237237024|1237237030|1237237024|0
229383|1237237030|1237237030|1237237030|1237237030
229382|1237237033|1237237030|1237237033|0
229380|1237237041|1237237019|1237237041|0
229381|1237237024|1237237030|1237237041|0
229382|1237237041|1237237030|1237237041|0
229383|1237237041|1237237041|1237237041|1237237041
229383|1237237041|1237237041|1237237041|0
229381|1237237024|1237237030|1237237041|1237237041
229378|1237237007|1237237044|1237237007|0
229380|1237237044|1237237019|1237237044|0
229382|1237237044|1237237030|1237237044|1237237044
229381|1237237049|1237237049|1237237049|0
229380|1237237059|1237237054|1237237059|0
229383|1237237059|1237237049|1237237059|1237237059
229378|1237237007|1237237064|1237237007|0
229379|1237237064|1237237019|1237237064|0

Figure 11: Condensed TimeKeeper Output

Next, the interesting inode numbers are extracted from the metadata archival output and each of them is used to query a database created from the logged VFS data. This action produces Figure 12. In this figure, the queried inode numbers are bolded with the corresponding dentry data following. Indented underneath the dentry data is information corresponding to the parent dentry. The dentry data consists of the filename, the file type, and the max of the mtime, atime, and ctime at the time the dentry is instantiated. It can be seen what filenames correspond to the unidentified inodes. Also in the case that an inode number is recycled, the filenames are shown for each instance that it is used.

By using the data from Figure 11 and Figure 12 together, without considering the known sequence of events, a reasonable scenario of what has taken place can be constructed. For example, it is shown in Figure 12 that inode 229378 is only associated with the filename

229377 1237237019
March-16-2009 Directory 1237236997
3900380 Directory Testing
229382 1237237044
.Test1.txt.swp Regular File 1237237030
229380 Directory Tests5
.Test1.txt.swp Regular File 1237237030
229380 Directory Tests5
229381 1237237049
Test2.txt Regular File 1237237049
229380 Directory Tests5
Test1.txt Regular File 1237237024
229380 Directory Tests5
229380 1237237059
Tests5 Directory 1237237019
229377 Directory March-16-2009
229383 1237237059
4913 Regular File 1237237041
229380 Directory Tests5
Test1.txt Regular File 1237237041
229380 Directory Tests5
.Test1.txt.swx Regular File 1237237030
229380 Directory Tests5
229378 1237237064
Test_short.sh Regular File 1237237007
229377 Directory March-16-2009
229379 1237237064
Times5.txt Regular File 1237237019
229377 Directory March-16-2009

Figure 12: Condensed Correlation Output

Test_short.sh. In Figure 11, it can be seen that this file is accessed repeatedly throughout the experiment at time intervals ranging from 5 to 15 seconds. Although the file content is unknown at this point, if it is assumed that Test_short.sh is indeed a shell script file, then this behavior is justified as the file is probably read from sequentially to fetch each command. This claim can be bolstered by the fact that during the process, only the last access timestamp of the inode is updated in the journal.

Another conclusion can be drawn by examining the inodes that have been deleted, 229381, 229382 and 229383. The dentry log output shows that these inodes were associated the files Test1.txt, .Test1.txt.swp, .Test1.txt.swx, and Test2.txt. The conclusion that can

be drawn is that these were backup files used by a text editor that eventually replaced the original file, Test1.txt, which apparently started out associated with inode 229381 before being reassigned to inode 229383 from which it was later deleted. In [21] and [56], the `vim` program was documented as exhibiting similar behavior to that mentioned above. This hypothesis can be verified by widening the scope of previous queries to show other inodes accessed during the same period in question. After correlation, this action reveals filenames specifically associated with the vim program.

Inode 229382 appears to have duplicate entries in that .Test1.txt.swp appears twice with the exact same timestamp and parent dentry information. This is not a typo, d_instantiate was actually called twice. This behavior suggests that the previous dentry was deleted and reinstantiated. To confirm this behavior, the d_drop function would have to be instrumented.

In Figure 12, the time that appears next to each of the file types is the MAC-time. That is to say, the mtime, atime, and ctime are all equivalent. This is caused by the experiment creating and deleting files fairly rapidly and the information only being recorded when a dentry is instantiated.

To prove the file system independent nature of dentry logging, this same experiment was executed on a NTFS partition with similar results. No TimeKeeper-like system was available to pull from the file system journal, or log file, in the NTFS experiment. In the results that were obtained, all of the files created and even VIM library files could be found. These results would be more meaningful if the MAC times were captured as each file was modified instead of at each inode instantiation point.

3.6.1 d_lookup Testing

In Figure 13, the output from the TimeKeeper database is shown when the previous experiment was repeated in the same environment with the d_lookup function instrumented instead. These results represent a small fraction of the total inode activity that was observed during the time period in question. Further narrowing was done based on the range of inodes that were seen in the output of the experiment log, shown in Figure 14. As the

```
inodeNum    mtime       atime       ctime       dtime
----------  ----------  ----------  ----------  ----------
3899509     1264044024  1264044023  1264044024  0
3899510     1264044019  1264044024  1264044019  0
3899511     1264044024  1264044024  1264044024  0
3899519     1264044024  1264044024  1264044024  0
3899519     1264044029  1264044024  1264044029  0
3899520     1264044029  1264044029  1264044029  0
3899510     1264044019  1264044034  1264044019  0
3899521     1264044037  1264044037  1264044037  0
3899522     1264044037  1264044037  1264044037  1264044037
3899521     1264044042  1264044037  1264044042  0
3899522     1264044053  1264044053  1264044053  0
3899519     1264044053  1264044024  1264044053  0
3899520     1264044029  1264044037  1264044053  0
3899521     1264044053  1264044037  1264044053  1264044053
3899522     1264044053  1264044058  1264044053  0
3899510     1264044019  1264044058  1264044019  0
3899519     1264044058  1264044024  1264044058  0
3899520     1264044058  1264044058  1264044058  0
3899510     1264044019  1264044063  1264044019  0
3899519     1264044058  1264044063  1264044058  0
3899511     1264044063  1264044024  1264044063  0
3899510     1264044019  1264044068  1264044019  0
3899519     1264044068  1264044063  1264044068  0
3899522     1264044068  1264044058  1264044068  1264044068
3899510     1264044019  1264044073  1264044019  0
```

Figure 13: Abridged d_lookup TimeKeeper output

d_lookup function is often called to check the dcache for the presence of a dentry, it is prone to failing in this task. This then results in control being handed over to d_instantiate.

When examining Figure 15, it becomes apparent that the correlation strategy used for d_instantiate produced logs will not work in this case. However, the absence of an inode information when the d_lookup is called, does not mean interesting information cannot be obtained. If d_lookup fails to find a dentry, the letter "f" is appended to the name and an inode number of -1 is assigned for the missing VFS structure. In these instances, the most important piece of information that can be obtained is the name of entry that is being queried. These values, shown in the third column of Figure 15, display all of the filenames

```
kevin@Random:~/Testing/January-20-2010$ cat Times5.txt
Wed 20 Jan 2010 10:20:24 PM EST Thu 21 Jan 2010 03:20:24 AM UTC-1264044024
total 16K
3899519 drwxr-xr-x 2 kevin kevin 4.0K 2010-01-20 22:20 .
3899509 drwxr-xr-x 3 kevin kevin 4.0K 2010-01-20 22:20 ..
3899522 -rw-r--r-- 1 kevin kevin   29 2010-01-20 22:20 Test1.txt
3899520 -rw-r--r-- 1 kevin kevin   29 2010-01-20 22:20 Test2.txt
Wed 20 Jan 2010 10:21:13 PM EST Thu 21 Jan 2010 03:21:13 AM UTC-1264044073
kevin@Random:~/Testing/January-20-2010$
```

Figure 14: Abridged d_lookup experiment log

return status	inode number	query	filename	parent inode number	parent filename
d_lookup-s	679843	15137	15137, 15137	1	/, /
d_lookup-f	-1	Test1.txt	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	$TMPDIR	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	0	v59176, v5917	557057	tmp, tmp
d_lookup-f	-1	.Test1.txt.swp	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	.Test1.txt.swx	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	4913	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	Test1.txt~	Tests5, Tests	3899509	January-20-2010,
d_lookup-f	-1	Test1.txt	Tests5, Tests	3899509	January-20-2010,
d_lookup-s	679847	15141	15141, 15141	1	/, /
d_lookup-f	-1	Test2.txt	Tests5, Tests	3899509	January-20-2010,
d_lookup-s	679868	15148	15148, 15148	1	/, /
d_lookup-s	679862	15144	15144, 15144	1	/, /

Figure 15: Abridged d_lookup log output

that are expected to be seen given the prior results from the d_instantiate experiment. In order to give these names some context, the dentry printing function is called on the parent dentry. Thus the query name does not match the filename for these instances. Although not as informative as the d_instantiate, definitive conclusions can be drawn as to the files that are being searched for. For example, from line 5 of Figure 15, it can be stated that the file .Test1.txt.swp was being looked up in the path "January-20-2010/Tests5/".

3.7 Performance

Table 4: Clean Kernel Compile Times

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Mean**	**St. Dev.**	**%**
Real	5942.42	5869.22	5858.34	5872.57	5864.32	5881.37	34.54	0.59
User	4875.69	4859.2	4861.08	4859.12	4851.74	4861.37	8.77	0.18
System	655.78	650.35	651.24	653.12	652.19	652.54	2.09	0.32

To measure the performance impact the dentry logging system has on a honeypot, version 2.6.26.5 of the Linux kernel was compiled several times in a clean (logging disabled) and a dirty (logging enabled) environment. The bash built-in `time` command was used to measure how long it took to complete this resource intensive task. The results are displayed

Table 5: Dirty Kernel Compile Times

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Mean**	**St. Dev.**	**%**
Real	6355.66	6338.3	6398.42	6365.82	6334.05	6358.45	25.79	0.41
User	4884.96	4884.81	4883.51	4888.31	4896.12	4887.54	5.11	0.1
System	687.55	678.56	683.68	683.23	682.62	683.13	3.2	0.47

in Tables 4 and 5. After five iterations, the standard deviation of each population was fairly low in comparison to the mean of the set. This can seen in the percent column. By comparing the means of the clean and dirty compile times using of the percent difference formula, Table 6 was produced. It should be noted that honeypots do not provide production computer system related services, but are instead used to gather security information to assist in enhancing the overall safety of the production system and its operating environment. With the proposed additional logging functionality enabled on a honeypot, there is a measured 8% performance penalty in order to gain increased visibility into system events. Furthermore as this is a proof of concept system, no effort was made to maximize efficiency, thus this performance impact could be reduced.

Table 6: Percent Differences

$$\%Diff = \frac{2*|x_1 - x_2|}{x_1 + x_2} * 100$$

	Differences	Percent Difference
Real	477.08	7.8
User	26.17	0.54
System	30.59	4.58

3.8 Futurework

Moving forward, more analysis regarding this technique and how it will fit together with other systems such as the TimeKeeper journal monitoring system should be performed. This would include experimentation and measurements to determine which function or combination of functions yield the best logging data. Furthermore, rapid file creation and deletion may not always be recorded in the journal due to higher level caching. The use of this technique with metadata archival should yield insight into these occurrences which can be leveraged to develop probability models about the effects of certain actions on the Ext3 journal.

Further experiments need to be conducted to determine the separate and cumulative affects that the dentry logger and metadata archiving systems have on the performance of a system. Given that the inode structure can be accessed from the dentry cache, it may be possible to completely replace metadata logging with dentry logging. Although this would create a great amount of data, dentry events can be viewed as atomic in nature

making the data a great candidate for inclusion in projects such as Zeitline [7]. Finally, the implementation of the techniques should be refined so that they work together in a less disjointed fashion. Once a more polished implementation is available the tools should be deployed in a large scale honeypot environment to collect realistic attack data.

3.9 Conclusions

This chapter presented a method for performing inode number to filename translations. The approach used takes advantage of the dcache in the Linux Virtual File System layer. This translation technique logs information in real-time, which is processed off-line and can be used in conjunction with metadata archived by other forensic tools. The benefits of this system are that it adds detail that would be very difficult to retrieve using the journal data alone, while providing the functionality of logging across different file system implementations. Also, by modifying the kernel code as opposed to using a kernel module, the dentry logging has become difficult to subvert. This avoids issues that can arise when the administrative user account cannot be fully trusted. By providing a richer dataset, this technique signifies a step forward toward the goal of providing additional means to detect and investigate anti-forensic attacks on honeypots.

CHAPTER IV

EXT4

4.1 Motiviation

The fact that Ext4 is becoming the default file system on popular Linux distributions, almost guarantees that it will be the subject of future digital forensic investigations. In this chapter an investigation of how the differences between Ext3 and Ext4 affect file system forensics is given. The new file system presents some unique challenges not only to digital forensics but to privacy in general which will be identified. Therefore, strides must be made in the open source forensic community for its support.

4.2 Introduction

As of the last half of 2009, Ext4 is the default file system for new installations of Ubuntu 9.10 [9], Fedora 11 [45], and openSuse 11.2 [36]. In fact, in the latest release of Fedora, Ext4 is used not only in the primary volume as the file system but in the boot partition as well [39]. Its predecessor, Ext3, is one of the most highly used Linux file systems but has certain limitations that the developers of the file system sought to address as the size of storage devices has and continues to increase.

Although Ext4 would seem like a relatively new file system, extensions to Ext3 have been proposed by its maintainers since at least 2002 [52]. For the most part, Ext3 can be seen as an extension of the previously popular Ext2. On the other hand, Ext4 can be best viewed as an evolution of the file system rather than just Ext3 with new features.

Ext4 is backward compatible with Ext3, but Ext3 is not forward compatible with Ext4. Unlike the nearly seamlessly transistions that can take place between Ext2 and Ext3, Ext4 is only partially supports the older file systems. This is due to the fact that some on-disk structures have been changed in the development of the new file system. Ext3 partitions can be mounted using Ext4. In this type of environment, older files will continue to use indirect block mapping while newly created files will use the larger inode structure and

extents. It is possible to obtain some of the benefits of Ext4 by mounting an older partition with out extent support. This scenario may deliver the benefits of the Ext4 block allocation algorithms, but without the use of extents the hybrid file system is somewhat handicapped. Due to the fact that some of the structures used by the older file systems and the new one are different, Ext4 cannot be mounted by Ext3.

Ext4 should not be thought of as a stopping point in Linux file system development. In [51], Tso explicitly states that the long term plan was to have a next generation filesystem for Linux with Btrfs being the best candidate. However due to Btrfs being in its early stages and the amount of time it takes truly develop a new file system from scratch, the decision was made to develop Ext4. This decision has the benefits of allowing developers to capitalize on the knowledge gained from the work previously done in the development of Ext3 while giving users the ability to upgrade their previously existing file system in steps. By starting from the Ext3 code base and extending it, developers planned for Ext4 to gain the capabilities of the previous version with the addition of new features in a relatively short period of time [34].

As file systems can be the most abundant source of digital information in a forensic investigation, the differences between Ext4 and Ext3 need to be examined and new methods to explore Ext4 formatted disk images must be devised. The SleuthKit (TSK) [14], which is based on the Coroner's Toolkit, is one of the most documented and trusted open source forensic toolkits. It is actively maintained and there are at least two extensions to the command-line tools of which it is comprised. The most well known is the Autopsy Forensic Browser which was developed by the creator of TSK, but PTK [16] provides a much more advanced interface to aid forensic analysis.

Interfaces aside, the core of TSK is a set of Unix-based tools executed at the terminal. It can be used to analyze FAT, NTFS, UFS, as well as the Ext2 and Ext3 file systems. As file systems advance the forensic tools used to analyze them must advance as well. Even if Ext4 is only a temporary stop on the way to Btrfs, its adoption by major players in the Linux community almost ensures that at some point the file system will fall under the scrutiny of an digital investigation. To that end, this chapter presents initial research findings which

are being used to develop a set of tools specifically for the forensic analysis of Ext4.

4.3 Background

Although Ext4 and its predecessors, Ext2 and Ext3, have some things in common there are differences that must be taken into account when considering the forensic aspects of these file systems. Most of the basics about these file systems have been discussed in Chapter 2. Section 2.1.1 focused on Ext3 while section 2.1.3 provided details about Ext4.

4.4 Hypothesis

As Ext4 is based upon the Ext3 file system, certain assumptions have been made. Since Ext3 zeros inode block pointers upon file deletion, one may assume that Ext4 would have similar behavior. This assertion stems from the fact that the extents, either the root of a tree or in-inode extents, are placed in the same location that the direct and indirect block pointers are stored in when considering the previous version of the inode structure. This may be a safe assumption, and gathering proof of this operation is but a simple step along the way toward testing a the larger hypothesis of this chapter that preallocation in Ext4 can lead to an unintentional violation of privacy under certain conditions.

When large sparse files are created in Ext4, the entire space for the file will be reserved. However, all of that area will not be initialized. The areas that are not immediately needed will be allocated using an extent that has a flag set denoting that it is holding free space. This saves time when huge files such as virtual machines are being created that actually have data located sparingly around an image file. However, a certain danger exists in the fact that when the VFS reads one of these place holding extents, it interprets it as representing zeros. In a sense, the VFS is fibbing. This chapter proposes that from the sparse file, actual data can be retrieved from files that have either been deleted or, in the case of defragmentation, moved to make room for the large file. Furthermore, if there are multiple users competing for space in the file system, the sparse file could possibly be allocated blocks that once belonged to another user. Depending upon the length of the lifespan of place holding extents, data from other users could possibly be retrieved from within the sparse file.

Finally it is theorized that the SleuthKit, a well accepted open source forensic toolkit, and such other forensic toolkits will return inaccurate data when analyzing the an Ext4 partition. This is assertion is made knowing that the Sleuthkit was designed for systems such as Ext2 and Ext3, however support for Ext4 will not be as simple as adding extra fields to various data structures to support 64bit file systems. The fact that the relationship between blockgroups and their associated metadata structures has been altered, means that certain assumptions made when designing the Sleuthkit will not hold true in Ext4. This means that data reported by the Sleuthkit may be inaccurate not only when dealing with extents but when processing group descriptors and other fundamental file system components.

4.5 Experimentation

The experiments conducted in this research made use of Fedora 11 and Ubuntu 9.10 installed inside of the Sun VirtualBox environment. Each virtual machine was mounted as a raw image such that it could be analyzed from an outside source with relative ease.

To show an instance where the Ext4 filesystem can give a false sense of information deletion to the user, a large file filled with null values was created in the same space where a plethora of small files containing duplicate information were freshly deleted. In particular, a small (500MB) Ext4 filesystem was created and filled to 100% utilization with 4KB files, each filled with the word "TEST" or with other known contents equaling exactly 4KB. This was done to eliminate any file slack space. The files were then unlinked (deleted) from the filesystem. Then, a large file of null values was created to fill the entirety of the available space using the `dd` command. The filesystem was then synchronized, unmounted, and searched for traces of the "TEST" message. Surprisingly, there were 22MB of data that remained untouched by the zeroing process. While inside of the file system, when the large file is searched, there isn't any of the TEST data. Hence, Ext4 tells us that the file is filled with zeros, when physically, there is actually data remaining on the device.

The second experiment attempted to exploit online defragmentation, a process allowing for smaller files to be moved to allow larger files to remain more contiguous. This could cause sensitive files to be moved without securely deleting the source information. If the

large file being written is sparse, then the sensitive data would not be overwritten. The problem foreseen by this is a scenario where a user would perform a secure wipe of the data, only to be undermined by the filesystem leaving a copy at an earlier location on the device. To perform this experiment, a new 500MB file image was formatted with Ext4, files were written to fill the device, some files were deleted, and a large file was written to try to force the filesystem to move files around. To fill the device, 4KB and 16KB files were written alternatively on the device until it was full. Then the 16KB files were deleted and the extents for all the 4KB files were dumped into a text file. Next, a file of zeros was written to fill the filesystem and the extents for all the 4KB files were dumped again and compared to the first file. The two dumps were identical, indicating that online defragmentation did not take place. Upon further investigation, it was discovered that the defragmentation feature was not included in the version of Ext4 installed with the operating systems used for the experiments. This security threat can not be fully analyzed until a stable implementation of the online defragmentation algorithm is incorporated.

4.6 Results

4.6.1 Data Pointer Zeroing

In Ext2, the recovery of a deleted file was not exactly simple, but it was possible given that the inode structure still contained pointers to the data blocks. Therefore, if a user was lucky enough not to reuse a particular inode nor any of its data blocks, they could reconstruct a file. Ext3 made this a little more difficult in that the data block pointers for the inode structure were zeroed during deletion. This practice made file carving necessary if one wanted to recover data. File carving is the practice of examining data blocks, usually unallocated, and applying various algorithms to determine if they contain files. It is analogous to examining a great deal of jigsaw puzzle pieces that are all cut into perfect squares, thereby only giving the puzzle solver the images on the pieces (data in the blocks) to determine which pieces belong together. The problem is exacerbated if multiple puzzles (files) are mixed together. Ext4 continues the practice of zeroing out the data information from an inode upon deletion.

```
Inode 438 is part of block group 0
        located at block 346, offset 0x0280
00056a80   a4 81 00 00 00 10 00 00   67 df 01 4b 67 df 01 4b
00056a90   67 df 01 4b 00 00 00 00   00 00 01 00 08 00 00 00
00056aa0   00 00 08 00 01 00 00 00   0a f3 01 00 04 00 00 00
00056ab0   00 00 00 00 00 00 00 00   04 00 00 00 a9 28 00 00
00056ac0   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
*
00056ae0   00 00 00 00 c5 84 58 dd   00 00 00 00 00 00 00 00
00056af0   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

Figure 16: Short Valid Extent Inode

Figure 16 shows the hexdump of an inode that was retrieved after the file system was completely filled. The state of the inode after deletion can be seen in Figure 17. When analyzing this output, we know that the inode is using extents and not direct blocks because at the fortieth byte the magic number for the extent header, 0xf30a[1], is present in both figures. When the inode is allocated, timestamps can be seen at byte offsets 8, 12, and 16. These are consistent with the timestamp locations in the second figure which has updated values for the modification, change and deletion times for the inode.

```
00056a80   a4 81 00 00 00 00 00 00   67 df 01 4b 18 f3 01 4b
00056a90   18 f3 01 4b 18 f3 01 4b   00 00 00 00 00 00 00 00
00056aa0   00 00 08 00 01 00 00 00   0a f3 00 00 04 00 00 00
00056ab0   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
*
00056ae0   00 00 00 00 c5 84 58 dd   00 00 00 00 00 00 00 00
00056af0   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

Figure 17: Short Deleted Extent Inode

As the file's size was 4096 bytes and the file system block size was 1024 bytes, the file consumes 4 blocks. The starting data block for the file is 10409 (0x28A9). These values may be found on the line beginning with 00056ab0 in Figure 16. After deletion, the entire line is zeroed, affectively removing any chance to easily recover data. Because of this behavior, data carving will continue to be necessary when performing the forensic analysis of an

[1]Note that in Figures 16 and 17 the values are stored in little endian format.

Ext4 file system. However, as the Ext4 block allocation schemes do their best to prevent file fragmentation, the data carving maybe more successful if it is initiated soon after the original file has been deleted. Some adjustments to current data carvers and further testing is necessary to confirm this suspicion.

4.6.2 Sleuthkit Testing

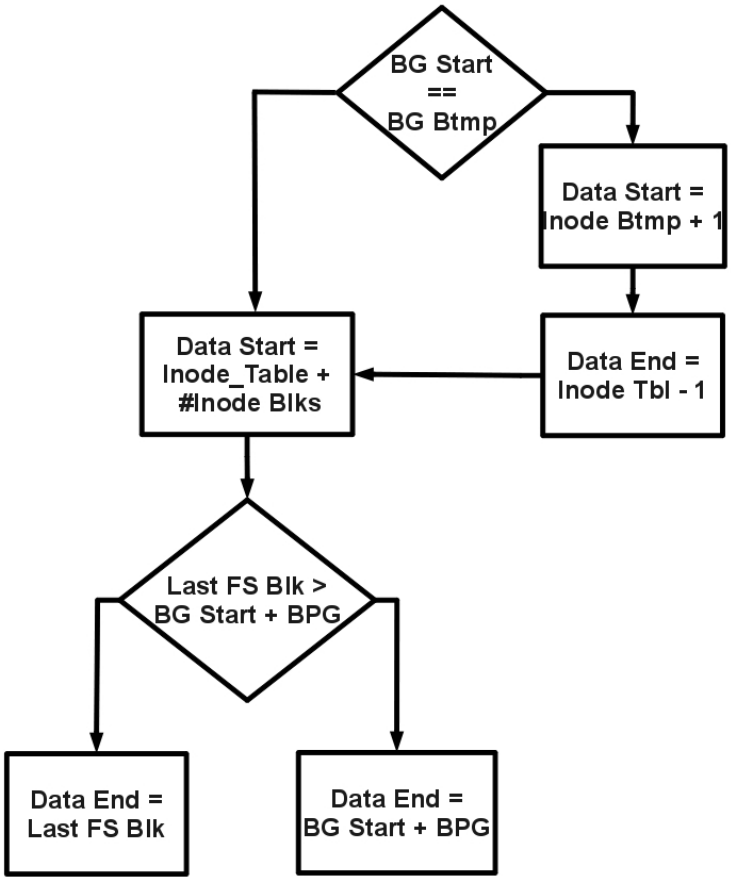

Figure 18: TSK Data Block Decision Chart

The Sleuthkit's `fsstat` command is able to process the block groups and exit normally when analyzing a non-64 bit Ext4 file system with 32-bit group descriptors. However, the current version of `fsstat` will provide inaccurate information. The algorithm the Sleuthkit uses to process block group descriptors is made for Ext2 superblocks, which have less features than an Ext4 superblock. When comparing the output of the `fsstat` command, Figure 19, to the output of the Ext4 version of the same command, shown in Figure 20, which was

prototyped as a product of this research, it can be seen that the incompatible features section for Ext4 has several more entries than that of its predecessor's output. Also note that the `fsstat` output displays every block group as containing a copy of the file system superblock. Although the use of extents is an important concept, here focus is specifically drawn to the Flex_BG feature. Flex_BG is enabled by default on fresh installs of Fedora 11 and Ubuntu 9.10. As was discussed in section 2.1.3.1 Flex_BG changes the layout of important metadata components inside of a file system.

An examination of TSK's `fsstat` source code revealed that it assumes if the start of a block group and its data block bitmap are not at the same block, the first block of that block group must contain a backup of the file system superblock. Flex_BG, along with virtual blockroups, break this assumption. With Ext4 filesystems, the majority of the time the datablock bitmap will not be the same as the start of the block group. In fact, the block bitmap is not in located in its block group the majority of the time. This also affects how data blocks are reported by the `fsstat` tool in TSK. If the block group base was equal block group bitmap number, then the data block range is calculated using the inode bitmap and inode table block numbers. Otherwise, the start is reported using the inode table number augmented by the number of blocks it takes to store the table. The last datablock is reported as either the last block mathematically or the last block in the filesystem if the final block group is only partially filled. The TSK decision making concerning data blocks is shown in Figure 18. In short, forensic researchers and toolkit designers must take virtual block groups into account and be wary of the output of current forensic tools.

4.6.3 Data persistence in sparse files

After filling a small Ext4 partition with small files, deleting them, and creating a large file with zero data, the next step was to attempt to retrieve the previous data. Although the inode extents were zeroed out and the file system reported all of the blocks belonging to the large file as zero, a hexdump of the unmounted file system image revealed otherwise. In Figure 21 several important pieces of data are displayed. At the top, the filenames, which are stored as data in directory structures, are stored. Then, in the middle starting at line

```
Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode,
Dir Index
InCompat Features: Filetype,
Read Only Compat Features: Sparse Super, Has Large
Files,

Group: 0:
  Inode Range: 1 - 8112
  Block Range: 0 - 32767
  Layout:
    Super Block: 0 - 0
    Group Descriptor Table: 1 - 1
    Data bitmap: 611 - 611
    Inode bitmap: 627 - 627
    Inode Table: 643 - 1149
    Data Blocks: 1150 - 32767
  Free Inodes: 22 (0%)
  Free Blocks: 3584 (10%)
  Total Directories: 4

Group: 2:
  Inode Range: 16225 - 24336
  Block Range: 65536 - 98303
  Layout:
    Super Block: 65536 - 65536
    Group Descriptor Table: 65537 - 65537
    Data bitmap: 613 - 613
    Inode bitmap: 629 - 629
    Inode Table: 1657 - 2163
    Data Blocks: 2164 - 98303
  Free Inodes: 24 (0%)
      Free Blocks: 3979 (12%)
      Total Directories: 688

Group: 4:
  Inode Range: 32449 - 40560
  Block Range: 131072 - 163839
  Layout:
    Super Block: 131072 - 131072
    Group Descriptor Table: 131073 - 131073
    Data bitmap: 615 - 615
    Inode bitmap: 631 - 631
    Inode Table: 2671 - 3177
    Data Blocks: 3178 - 163839
  Free Inodes: 0 (0%)
  Free Blocks: 3581 (10%)
  Total Directories: 1491
```

Figure 19: TSK fsstat output

```
***Compatible Flags***
incompat_flags: 3C
Has Journal          Extended Inode Attributes
FS resizing (Resize Inode)    Directory Indexing
***Incompatible Flags***
incompat_flags: 242
Directories have file types     Uses Extents
Flex BG
***Read Only Compatible Flags***
ro_flags: 7B
Sparse Super         Large Files
Huge Files           Group Descriptor Checksums
Dir_N_Link           Extra ISIZE

Flex_Group_Size:
          Size in Bytes: 2147483648,
          Size in Groups: 16,
          # of Flex Groups: 5
Group 0:
          Group 0, has a superblock
          SuperBlock at, 0
          GD Block at, 1
          GD Reserved at 2-610
          Block group flags: 04
          Inode Table initialized to Zero
  Blocks Range: 0 - 32767
          Block Bitmap: 611
          Inode Bitmap: 627 (offset 16)
          Inode Table: 643 - 1149
          Directories: 4
          Free Blocks: 3584
          Free Inodes: 22
          Inode table unused: 0
Group 2:
          Block group flags: 04
          Inode Table initialized to Zero
  Blocks Range: 65536 - 98303
          Block Bitmap: 613
          Inode Bitmap: 629 (offset 16)
          Inode Table: 1657 - 2163
          Data Blocks: 65536 - 98303
          Directories: 688
          Free Blocks: 3979
          Free Inodes: 24
          Inode table unused: 0
Group 4:
          Block group flags: 04
          Inode Table initialized to Zero
  Blocks Range: 131072 - 163839
          Block Bitmap: 615
          Inode Bitmap: 631 (offset 16)
          Inode Table: 2671 - 3177
          Data Blocks: 131072 - 163839
          Directories: 1491
          Free Blocks: 3581
          Free Inodes: 0
          Inode table unused: 0
```

Figure 20: Ext4 fsstat output

0x00880410, the contents of the script that was actually used to repeatedly make copies is displayed. Finally, at the bottom the start of the file which was copied is shown.

This one snapshot captures a couple of layers of abstraction as the filenames are the content of the parent directory and content of the file in that directory are displayed just below it. This situation is exacerbated by the fact that the root account was used to create the files originally, but a regular user account was used to create the sparse file. This

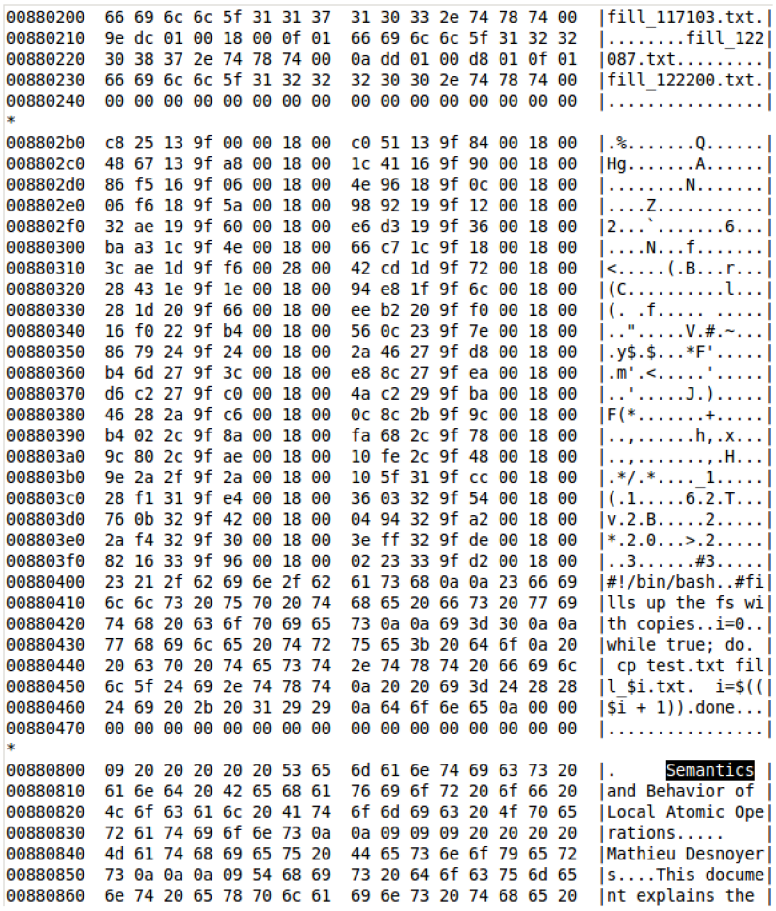

Figure 21: Embedded File Remnants

means that under the correct circumstances, any users' data may be covered by a sparse file, thereby making it a very interesting target in terms of file system forensics.

4.7 Future work

4.7.1 Online Defragmentation Testing

Once the online defragmentation has been stabilized, testing should be done regarding that feature with it enabled. If a large sparse file displaces sensitive data that is later deleted, this could result in a security incident as remnants of the file could lay just underneath the sparse file.

4.7.2 Timestamps

One of the challenges of event reconstruction in the Ext3 file system has been the resolution at which events are recorded. Considering the processing power of modern computers, many different actions can take place in the span of a second. While in Ext3 they would all appear to have happened simultaneously, Ext4 offers the promise of information necessary to correctly sequence these events as the new larger inode structure has nanosecond resolution. Also, the insertion of the creation timestamp into the inode structure will serve as another source of forensic evidence.

4.7.3 Ext4 and TSK

While this research displays some of the shortcomings that the current incarnation of TSK has with regards to Ext4, it must be noted that TSK was not designed for this file system. Therefore this research serves as a starting point which should be completed when a full set of open source Ext4 forensic tools similar to those of the TSK are made available. Furthermore, one option would be the integration of the work done here into a future release of the Sleuthkit. This approach would yeild the following benefits:

1. Open Source

   TSK is an open source toolkit and studying its source code has aided the progression of this research. By combining the work done here with TSK, the open source the forensic community will have a chance to add on, correct, and make the code more efficient.

2. Well Supported

   TSK is well supported with an active developers mailing list. Thas has probably resulted in an increased lifespan as overtime more file systems and different image formats have become supported. By aligning with this community, the code will be maintained and benefit from a fairly well established library of functions.

3. Wide Spread Adoption

   As opposed to releasing the code into the wild and hoping that it catches on by being

incorporated in popular forensic distribution releases, joining a project that is already well known will increase its chances of being adopted by a wide number of digital forensic practitioners.

4.7.4 Sparse File Analysis

More research must be done into the likelihood of sparse files overwriting either the data meant for deletion or the data of other users. Although it has been shown that when space is limited the deleted data of a higher privileged user may become compromised, it is unknown how often these circumstances will occur. To this end, the development of a method to rapidly analyze large sparse files for sensitive remnants would be of great use.

4.8 Conclusions

Ext4 has become the file system of choice on several popular Linux distributions. It offers many benefits over its predecessors, but is only intended to be a stopping point until Btrfs stabilizes. Some of the intrinsic behaviors in Ext4 present new challenges to the forensic and security community while also providing better sources of data. One particular behavior of Ext4 is how it handles large sparse files such as databases, drive images, and raw virtual machine files.

The preallocation feature reserves space for these files, while not fully initialing all of the extents. This supposedly leaves data, which was not being used on disk anyway. When blocks belonging to the uninitialized extents are read from the disk, the VFS interprets them as zeros although there may actually be valid file data underneath the sparse files. Thus the file system provides and inaccurate picture to the user with respect to contents on the disk in order to make gains in efficiency.

The upshot of this behavior is that when the file system becomes congested with users competing for space, what one user thinks he has deleted may still actually be on the disk incorporated into an uninitialized section of another user's sparse file. This situation can be further exacerbated by online defragmentation as a copy of a user's file could exist beneath a sparse file unbeknownst to that user. Thus the forensic community must not only develop methods to analyze Ext4 by taking advantage of features such as high resolution

timestamps, but also consider sparse files and other uninitialized structures as possible sources of information.

CHAPTER V

FUTURE WORK

As most of the ideas for future directions of the individual components presented in this thesis are covered in sections 2.7, 3.8, and 4.7; this chapter discusses future directions for the research in its entirety.

5.1 Visualization

Through experimentation it has been observed that archival of modification, access, and change times produces a great amount of information. Using carefully constructed query statements, a database consisting of these times may yield useful results, but this is not guaranteed. A major problem is that many inodes are accessed in the regular (non-malicious) use of a system. One reason for these accesses is these inodes could potentially hold shared libraries used by many programs. This makes the detection of anti-forensic techniques difficult. This difficulty is further compounded by the growth of modern operating systems, the applications that are commonly bundled with them, and the ever increasing sizes of modern hard disks.

Visual components would aid in the examination of data produced by archival components. This would reduce the effort expended by the forensic analyst. Instead of finding an anomaly among hundreds to thousands of lines of text, the task can be reduced to finding an unexpected trend in a visual representation of the archived data. The theorized visual component is not meant to completely replace the process of performing a detailed forensic investigation. It is simply an aid with the purpose of reducing the amount of data that an investigator may have to go through by drawing his or her attention to a particular area or time period.

5.2 Abstracted Query Language

Just as filenames have certain benefits over dealing with inode numbers directly, a filter language could provide similar advantages. The amount of data generated in the day to day functions performed by a modern operating system can be overwhelming. When attempting to comprehend what has taken place, an analyst does not need to be encumbered with the underlying details of complex data structures. Instead, an interface that rarely changes but provides them with a great deal of flexibility and functionality would provide more assistance. To this end, a standardized temporal query language would be a great addition to this research. In time, this temporal query language can become as useful in the establishment of time based events, as packet filter languages are when performing packet inspection. It would also facilitate rapid analysis and comprehension of the data.

The storing of archived data in databases and the construction of SQL queries is a valid method of analysis for retrieval of detailed information about the events that have transpired. The drawback is that this limits the framework and the analyst to a language with which they may not be familiar. Should the structure of the underlying database change, the SQL queries needed to extract the same data would be affected. The filter language will allow the user of tools based on the framework to employ a set of standard commands without having to know the details of implementation. The language will also permit analyst to comb through the volumes of data without having to construct complicated query statements. Furthermore, the development of a standardized temporal query language will allow the underlying components of the framework to be modified and updated. This layer of abstraction would allow further growth and development of the forensic framework.

5.3 Probabalistic Forensics

Further research should be conducted in the area deemed probabilistic forensics as presented in [56]. By developing and deploying systems which inspect memory structures and can determine what affect certain actions have on memory and on a file system, a level of confidence can be obtained as to what has transpired on a particular system. For example, many programs written in C will share certain libraries and thus access the same

files. However, if it is assumed that each program will possess behavior unique to itself, then there exists an opportunity to build probability models about which programs have been executed. Furthermore, different versions of the same program could possibly be differentiated. Although it may currently be possible to say with confidence that a series of events has transpired on a target computer, this particular line of research could lead to quantifiable measurements.

CHAPTER VI

CONCLUSION

In this thesis, the theory that additions to the operating system will yield better evidence for the forensic analysis of honeypots has been explored. To this end, mechanisms that have traditionally been used for other reasons have been thoroughly studied and adapted for logging purposes. In each chapter, a theory was presented followed by an explanation of a prototype, tests, and results to determine the viability of the theory. The fundamental assertion under which most of the research was conducted states that because honeypots differ from production systems, the forensic techniques traditionally used in enterprise environments do not adequately cover these reconnaissance purposed systems.

The value of a honeypot is only realized after it has been compromised, and that value can be increased if the attacker elevates their privilege level to that of an administrator. These actions allow honeypot researchers to examine the techniques with which an assailant gains access to systems and observe how compromised systems are being used. Furthermore, because honeypots do not host legitimate user activity, it is easier to discover malicious actions and keep audit logs for long periods of time. With this set of circumstances in mind, honeynet administrators cannot afford to entrust the administrative account with the ability to disable logging mechanisms.

When conducting the forensic investigation of a file system, the value of using file timestamps to construct a timeline of events should not be underestimated. However, because this source of information can be very important, there are documented techniques for altering timestamps. Chapter 2 detailed the TimeKeeper technique which uses the Ext3 journal as a source of information. By default, the journaling system records file metadata to the journal area of the disk before copying it over to the actual file system. This mechanism ensures that if the file system is used, the inode structure, and the thereby the metadata times, can be gathered from the journal. To test this theory, a prototype was constructed

and a brief test was conducted. The results displayed a history of timestamps about inodes that otherwise could have been considered unrelated. This shows that keeping a history of metadata times makes the usage of timestamps in an investigation more robust against tampering and that the file system journal is an excellent place to capture this information.

Although the TimeKeeper technique was a success, that line of research revealed another issue in the area of digital forensics - inode number to filename translation. The journaling system records information at block resolution and by default records metadata blocks. The problem that arises from this behavior is that filenames are actually stored as data the portion of a directory inode. Furthermore, filename to inode number translation is unidirectional. Although the data from a directory inode may be journaled, the process of capturing this data becomes very involved. For example, the monitoring system must process every inode it detects, determine which inodes are directory inodes, locate the data blocks from associated with those inodes, then parse the contents of those blocks. In Chapter 3, a more elegant and file system independent solution to this issue was covered. The technique makes use of the dentry cache, a part of the Virtual File System, in the Linux kernel. By making changes to the operating system instead of using a kernel module, the logging cannot be easily circumvented by an administrative account. This meets constraints imposed by operating in honeypot environments while yielding information that provides a history of what filenames have been associated with an inode number. This set of data can either be used independently for translation in a historical context or correlated with other data sets.

As time progresses, new technologies will continue to be introduced that present different challenges to the digital forensics community. Because honeypots are suppose to resemble production systems, these changes also have an impact on honeypot forensics. Since Ext4 has been adopted as the default file system by several major Linux distributions, it has an influence on both production and honeypot environments. Chapter 4 presented the forensic implications of the architectural differences between the Ext4 and Ext3 file systems. It was shown that because the layout of important file system metadata structures are different, current forensic toolkits can provide an analyst with inaccurate information. Experiments

also showed that because Ext4 uses preallocation, under certain circumstances large sparse files can contain remnants of older files that belonged to the owner of the sparse file or another user altogether. These issues must be further explored to ascertain how often this behavior can occur and whether methods can be devised in which this can be addressed.

This research has examined different levels of abstraction with respect to file system forensics. At the lowest layer, file systems where analyzed in detail. At a higher level, abstractions which file systems present to the Linux kernel were analyzed. Although techniques must be developed to analyze the on-disk structures of current file systems after an intrusion, audit logs gathered during an intrusion can be critical when conducting an investigation. This research has presented three techniques that can be used in honeypot forensics to provide additional detail about attacks. While the TimeKeeper and Dentry Logging prototypes can be deployed while the system is active, the Ext4 toolkit prototypes can be used after a compromise.

REFERENCES

[1] Access Data Corporation, "Forensic toolkit." http://www.accessdata.com/common/pagedetail.aspx?PageCode=ftk2test, 2007. Last Accessed: 11-19-2007.

[2] Asrigo, K., Litty, L., and Lie, D., "Using vmm-based sensor to monitor honeypots," in *Proceedings of the 2nd ACM/USENIX International Conference on Virtual Execution Environments*, pp. 13–23, 2006.

[3] Barik, M. S., Gupta, G., Sinha, S., Misra, A., and Mazumdar, C., "An efficient technique for enhancing forensic capabilities of ext2 file system," in *Digital Forensics Research Workshop*, 2007.

[4] Berinato, S., "How online criminals make themselves tough to find, near impossible to nab." http://www.cio.com/article/114550, May 2007. Published 05-31-2007, Last Accessed: 8-18-2009.

[5] Bovet, D. and Cesati, M., *Understanding the Linux Kernel, Third Edition*. Sebastopol, CA: O'Reilly Media, Inc., 3 ed., November 2005.

[6] Buchholz, F. and Falk, C., "Zeitline." http://sourceforge.net/projects/zeitline/. Last Accessed: 01-19-2010.

[7] Buchholz, F. and Falk, C., "Design and implementation of zeitline: a forensic timeline editor," in *Digital Forensics Research Workshop (DFRWS)*, 2005.

[8] Caloyannides, M., "Forensics Is So "Yesterday"," *Security & Privacy, IEEE*, vol. 7, pp. 18–25, March-April 2009.

[9] Canonical Ltd., "9.10 Technnical Overview — Ubuntu." http://www.ubuntu.com/testing/karmic/alpha6, 2009. Last Accessed: 10-5-2009.

[10] Cao, M., Tso, T. Y., Pulavarty, B., Bhattacharya, S., Dilger, A., and Tomas, A., "State of the art: Where we are with the ext3 filesystem," in *Proceedings of the Ottawa Linux Symposium (OLS)*, pp. 69–96, 2005.

[11] Carrier, B., "Digital Forensics Works," *Security & Privacy, IEEE*, vol. 7, pp. 26–29, March-April 2009.

[12] Carrier, B., "An investigator's guide to file system internals," in *FIRST Conference on Computer Security, Incident Handling & Response*, 2002. www.first.org/events/progconf/2002/d1-02-carrier-slides.pdf Last Accessed: 11-16-2007.

[13] Carrier, B., *File System Forensic Analysis*. Upper Saddle River, NJ: Pearson Education, Inc., 2005.

[14] Carrier, B., "The sleuth kit & autopsy browser." http://www.sleuthkit.org, 2007. Last Accessed: 11-18-2009.

[15] Das, S., Chattopadhayay, A., Kalyani, D. K., and Saha, M., "File-system intrusion detection by preserving mac dts: a loadable kernel module based approach for linux kernel 2.6.x," in *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, (New York, NY, USA), pp. 1–6, ACM, 2009.

[16] DFLabs, "PTK an alternative computer forensic framework." `http://ptk.dflabs.com/overview.php`. Last Accessed: 11-18-2009.

[17] Dornseif, M., Holz, T., and Klein, C. N., "Nosebreak - attacking honeynets," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pp. 123–129, 2004.

[18] e-fense, "Helix." `http://www.e-fense.com/helix/`. Last Accessed: 09-22-2009.

[19] Eckstein, K., "Forensics for advanced unix file systems," in *IEEE/USMA Information Assurance Workshop*, pp. 377–385, 2004.

[20] Eckstein, K. and Jahnke, M., "Data hiding in journaling file systems," in *Digital Forensic Research Workshop (DFRWS)*, pp. 1–8, 2005.

[21] Fairbanks, K. D., Lee, C. P., Xia, Y. H., and Owen, H. L., "Timekeeper: A metadata archiving method for honeypot forensics," in *Information Assurance and Security Workshop IAW '07, IEEE SMC*, pp. 114–118, 2007.

[22] Farmer, D. and Venema, W., "The coroner's toolkit." `http://www.porcupine.org/forensics/tct.html`, 1998. Last Accessed: 11-16-2007.

[23] Farmer, D. and Venema, W., *Forensic Discovery*. Upper Saddle River, NJ: Addison-Wesley Professional, 2004.

[24] Garfinkel, S., "Anti-forensics: Techniques, detection and countermeasures," in *The 2nd International Conference on i-Warfare and Security (ICIW)*, pp. 77–84, 2007.

[25] Garfinkel, T., "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, pp. 163–157, February February 2003.

[26] Guidance Software Inc., "EnCase Forensic." `http://www.guidancesoftware.com/products/ef_index.asp`, 2007. Last Accessed: 11-18-2007.

[27] Hay, B., Nance, K., and Bishop, M., "Live Analysis: Progress and Challenges," *Security & Privacy, IEEE*, vol. 7, pp. 30–37, March-April 2009.

[28] Holz, T. and Raynal, F., "Detecting honeypots and other suspicious environments," in *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, pp. 29–36, 2005.

[29] Kleiman, S. R., "Vnodes: An architecture for multiple file system types in sun unix," tech. rep., Sun Microsystems, 1985.

[30] Kumar, A., Cao, M., Santos, J., and Dilger, A., "Ext4 block and inode allocator improvements," in *Proceedings of the Linux Symposium*, vol. 1, pp. 263–274, July 2008.

[31] Love, R., *Linux Kernel Development*. Indianapolis, IN: Novell Press, 2nd ed., 2005.

[32] Love, R., "Kernel korner: Intro to inotify," *Linux Journal*, vol. 139, no. 139, p. 8, Nov. 2005.

[33] Manes, G. and Downing, E., "Overview of licensing and legal issues for digital forensic investigators," *Security & Privacy, IEEE*, vol. 7, pp. 45–48, March-April 2009.

[34] Mathur, A., Cao, M., Battacharya, S., Dilger, A., Tomas, A., and Vivier, L., "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, vol. 2, pp. 21–33, June 2007.

[35] Metasploit Project LLC., "TimeStomp." `http://www.metasploit.com/projects/antiforensics/`, 2005. Last Accessed: 11-16-2007.

[36] Novell, Inc., "Opensuse 11.2." `http://en.opensuse.org/OpenSUSE_11.2`, 2009. Last Accessed: 11-17-2009.

[37] Peisert, S., Karin, S., Bishop, M., and Marzullo, K., "Principles-driven forensic analysis," in *NSPW '05: Proceedings of the 2005 workshop on New security paradigms*, pp. 85–93, 2005.

[38] Prosise, C. and Mandia, K., *Incident Response: Investigating Computer Crime*. Berkeley, CA; London: Osborne/McGraw-Hill, 2001. Includes index.

[39] Red Hat, Inc., "Release Notes for Fedora 12." `http://docs.fedoraproject.org/release-notes/f12/en-US/html-single/`, 2009. Last Accessed: 11-17-2009.

[40] Reith, M., Carr, C., and Gunsch, G., "An examination of digital forensic models," *International Journal of Digital Evidence*, vol. 1, no. 3, pp. 1–12, Fall 2002.

[41] Richard, G. G. and Roussev, V., "Scalpel: A frugal, high performance file carver," in *2005 Digital Forensic Research Workshop (DFRWS)*, 2005.

[42] Sato, T., "ext4 online defragmentation," in *Proceedings of the Linux Symposium*, vol. 2, pp. 179–186, June 2007.

[43] Schneider, F. B., "Accountability for perfection," *Security & Privacy, IEEE*, vol. 7, pp. 3–4, March-April 2009.

[44] Siever, E., *Linux in a Nutshell*. Sebastopol, CA: O'Reilly & Associates, Inc., 2 ed., 1999.

[45] The Fedora Project: Nalley, D., "Release notes for fedora 11: File systems." `http://docs.fedoraproject.org/release-notes/f11/en-US/`, 2009. Last Accessed: 10-5-2009.

[46] The Grugq, "Defeating Forensic Analysis on Unix," *Phrack Magazine*, vol. 0x0b, p. 0x06, July 2002. Released: 07-28-2002, `http://www.phrack.org/issues.html?issue=59&id=6#article`, Last Accessed: 09-21-2009.

[47] The Honeynet Project, *Know Your Enemy: Revealing the security tools, tactics, and motives of the blackhat community*. Boston, MA: Addison Wesley, 2002.

[48] The Honeynet Project in *Know Your Enemy: Sebek*, pp. 1–21, 2003.

[49] The Honeynet Project, "Know your enemy: Sebek," in *Technical Paper*, pp. 1–21, 2003 November 17th.

[50] The Honeynet Project, "Tools for honepots." `http://www.honeynet.org/tools/index.html`, 2006. Last Accessed: 11-16-2007.

[51] Tso, T., "RE:[RFC] Btrfs mainline plans." `http://thread.gmane.org/gmane.linux.filesystems/26246/focus=26492`, 2008. Last Accessed: 11-17-2009.

[52] Tso, T. and Tweedie, S., "Planned extensions to the linux ext2/3 filesystem," in *USENIX Technical Conference*, 2002.

[53] Ubuntu Bugs, "ext4 defrag / defragment tool in jaunty -include." `https://bugs.launchpad.net/ubuntu/+source/e2fsprogs/+bug/321528`, 2009. Last Accessed: 11-18-2009.

[54] United States Air Force Office of Special Investigations and The Center for Information Systems Security Studies and Research., "Foremost." `http://foremost.sourceforge.net/`, 2007. Last Accessed: 11-24-2007.

[55] Xia, Y., Fairbanks, K., and Owen, H., "Establishing trust in black-box programs," in *IEEE SouthEast Conference*, pp. 462–465, 2007.

[56] Xia, Y., Fairbanks, K., and Owen, H., "A program behavior matching architecture for probabilistic file system forensics," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 4–13, 2008.

PUBLICATIONS

- Sean Sanders, Kevin Fairbanks, Sahitya Jampana, Henry Owen III, "Visual Network Traffic Classification Using Multi-Dimensional Piecewise Polynomial Models," IEEE SoutheastCon, 2010. March 2010. Accepted.
- Kevin D. Fairbanks, Ying H. Xia, Henry L. Owen III, "A Method for Historical Ext3 Inode to Filename Translation on Honeypots," 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), International Workshop on Computer Forensics in Software Engineering (CFSE) 2009. vol. 2, pp. 392-397.
- Kevin Fairbanks, Kishore Atreya, Henry Owen. "BlackBerry IPD Parsing for Open Source Forensics." IEEE SoutheastCon, 2009. March 2009. pp. 195-199.
- Ying Xia, Kevin Fairbanks, Henry Owen. "Visual Analysis of Program Flow Data with Data Propagation." Proceedings of the 5th international workshop on Visualization for Computer Security. Cambridge, MA. September 2008, pp. 26-35.
- Ying Xia, Kevin Fairbanks, Henry Owen. "A Program Behavior Matching Architecture for Probabilistic File System Forensics." ACM SIGOPS Operating Systems Review Special Issue on Computer Forensics. Vol. 42, Iss. 3. April 2008, pp 4-13.
- Kevin D. Fairbanks, Christopher P. Lee, Ying H. Xia, Henry L. Owen III. "TimeKeeper: A Metadata Archiving Method for Honeypot Forensics." 8th Annual IEEE SMC Information Assurance Workshop. West Point, NY. 20-22 June 2007, pp. 114-118.
- Ying Xia, Kevin Fairbanks, Henry Owen. "Establishing trust in black-box programs." IEEE SoutheastCon, 2007. March 2007, pp. 462-465.

VITA

KEVIN D. FAIRBANKS

Kevin D. Fairbanks was born in Birmingham, Alabama in 1983. In May of 2005, he received a B.S. in Electrical Engineering with a concentration in Computers from Tennessee State University. In May of 2007, he received a M.S. in E.C.E. from the School of Electrical and Computer Engineering at the Georgia Institute of Technology. On March 17th, 2010 Kevin successfully defended his thesis. His research interests include digital forensics, computer security, network security, security visualization, reverse engineering, and malware analysis.