12-20-2009

Advanced Techniques for Improving the Efficacy of Digital Forensics Investigations

Lodovico Marziale
University of New Orleans

Advanced Techniques for Improving the Efficacy of Digital Forensics Investigations

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by

Lodovico Marziale III

B.S. University of New Orleans, 1999
M.S. University of New Orleans, 2006

December 2009

Funding

This work was supported in part by the National Science Foundation under grant # CNS–0627226.

Acknowledgements

I would like to express my heartfelt gratitude to:

- my advisor, Dr. Golden G. Richard III. His inspiration and guidance were of critical importance in seeing me through this arduous task.

- Dr. Vassil Roussev, Dr. Jaime Nino, Dr. Juliette Ioup, and Dr. Edit Bourgeois for being on my dissertation committee.

- Dr. Mahdi Andelguerfi, without whose assistance I might never have started my graduate studies.

- all of my co-researchers for their contributions to the papers on which this research was based: Dr. Golden G. Richard III, Dr. Vassil Roussev, Dr. Loren Schwiebert, Santhi Sri Movva, Andrew Case and Andrew Cristina.

- my friends and family, for somehow managing not to give up on me, even if it was just barely.

Table of Contents

List of Figures

List of Tables

Abstract

Digital forensics is the science concerned with discovering, preserving, and analyzing evidence on digital devices. The intent is to be able to determine what events have taken place, when they occurred, who performed them, and how they were performed. In order for an investigation to be effective, it must exhibit several characteristics. The results produced must be *reliable*, or else the theory of events based on the results will be flawed. The investigation must be *comprehensive*, meaning that it must analyze all targets which may contain evidence of forensic interest. Since any investigation must be performed within the constraints of available time, storage, manpower, and computation, investigative techniques must be *efficient*. Finally, an investigation must provide a *coherent* view of the events under question using the evidence gathered. Unfortunately the set of currently available tools and techniques used in digital forensic investigations does a poor job of supporting these characteristics. Many tools used contain bugs which generate inaccurate results; there are many types of devices and data for which no analysis techniques exist; most existing tools are woefully inefficient, failing to take advantage of modern hardware; and the task of aggregating data into a coherent picture of events is largely left to the investigator to perform manually. To remedy this situation, we developed a set of techniques to facilitate more effective investigations. To improve reliability, we developed the Forensic Discovery Auditing Module, a mechanism for auditing and enforcing controls on accesses to evidence. To improve comprehensiveness, we developed ramparser, a tool for deep parsing of Linux RAM images, which provides previously inaccessible data on the live state of a machine. To improve efficiency, we developed a set of performance optimizations, and applied them to the Scalpel file carver, creating order of magnitude improvements to processing speed and storage requirements. Last, to facilitate more coherent investigations, we developed the Forensic Automated Coherence Engine, which generates a high-level view of a system from the data generated by low-level forensics tools. Together, these techniques significantly improve the effectiveness of digital forensic investigations conducted using them.

Keywords: Computer Forensics, Digital Forensics, Digital Investigation, File Carving, Live Forensics, RAM Forensics, Forensic Discovery, Forensic Discovery Auditing

Chapter 1: Introduction

1.1 Motivation

As the rate of technological advance continues to increase, science and society are forced to keep up. This is especially true with respect to the effects of the rise of the use of computers in society. Social networking software, like Facebook and Twitter continue to change the way people communicate. The same can be said for voice over IP telephony, a la Skype, and peer-to-peer file sharing networks. The pervasiveness of the Internet in general has made computers an integral part of everyday life. Email has been the de facto standard for business communication for some time, and digital phones are quickly rendering land line home phones obsolete. Debit cards are well on their way to replacing cash as the de facto currency. The combination of RFID and GPS technologies in widespread use allow real-time tracking of just about anything. But with all of the benefits of this increased use of technology comes a corresponding increase in its potential for abuse. The social networking phenomenon and RFID and GPS are rife with possibilities for privacy invasion. IP telephony opens up the possibility for anyone with Internet access to intercept anyone else's phone calls. The rise of E-commerce and usage of electronic payments has helped make identity theft and white collar financial crime a lucrative career for the less-than-scrupulous. Nearly anonymous peer-to-peer file sharing also allows child pornographers to share media with one another. Clearly, criminal activities have seen the benefits of ubiquitous computing as surely as more benign activities have. It is this fact that demands that there exist effective techniques for performing investigations of digital media.

1.2 Characteristics of Effective Digital Forensic Investigation

The science of Digital Forensics, a relatively new discipline, deals with discovery of evidence on digital devices in order to answer several questions including: what events have transpired?, when did they transpire?, who was responsible?, and what was the mechanism by which the events occurred? The set of devices to investigate encompasses a wide range, including computers (from smartphones to data centers), PDAs, game consoles, RFID devices and other types of digital equipment. Investigations proceed in four phases: digital media is collected, the data is examined, analysis of the data attempts to answer the questions under investigation given the evidence, and a report is generated. The ability to effectively conduct investigations is imperative, because in many cases the stakes can be high (life, liberty). There are many characteristics of effective digital forensic investigations; however this work focuses on a specific core set of four: an effective digital forensic investigation must be reliable, comprehensive, efficient, and coherent. In the following sections we discuss each of these, but first we define an important concept: digital evidence. From Carrier [1], "*digital evidence* is data that supports or refutes a hypothesis that was formulated during the investigation. This is a general notion of evidence and may include data that may not be court admissible because it was not properly or legally acquired."

1.2.1 Reliable

We say an investigation is reliable if the evidence generated is accurate and free from tampering, whether malicious or accidental. In the practical sense, in order for the results of an investigation to be considered reliable, the investigator must be able to show that the evidence recovered was handled properly - nothing has been added to, removed from, or changed on the original media. This is achieved in part in standard investigations with the use of evidence bags. These sealed, tamper-evident bags protect the evidence they contain, facilitate tracking the chain of custody of the items, and hold information about the objects contained. These functions are important, as evidence must pass admissibility tests, whether legal or just common sense, in order to be used. Digital evidence must adhere to similar standards of admissibility, but instead of being contained in physical bags, it is stored in files on disk.

1.2.2 Comprehensive

We say that an investigation is comprehensive if its analysis includes all potential items of interest. This may be impossible to achieve completely since resources dedicated to any investigation are finite, but a comprehensive investigation should analyze as many of the potentially interesting targets as possible. In a physical investigation this could mean searching the entire scene of the crime, or interviewing witnesses and associates of the suspects and victims. In the digital arena, where we cannot directly view the artifacts under investigation, this means using established tools and techniques to analyze digital systems, and, for each system, its many subsystems – file access and event timelines, physical media, system logs, installed programs, running programs, open network connections and files, physical RAM, game console devices, encrypted VOIP, registry settings, or search of audio, video files, large storage arrays, just to name a few. Only a comprehensive investigation can hope to come to the most accurate conclusions.

1.2.3 Efficient

We say an investigation is efficient if it maximizes the use of constrained resources, which can include processing power, data storage, time, and manpower. Digital systems are extremely complex creatures. Effective investigation requires analysis of digital artifacts at several layers such that an investigator can make sense of the workings of the system as a whole. As many types of data are not plainly displayed to an investigator (deleted files, file fragments, password protected and encrypted files, proprietary file formats), accessing this data requires a large number of tools and techniques, each of which require some use of resources. Of all of the possibly relevant objects, the investigator must pick and choose which to focus his limited resources on, whether these resources are computation, storage, or time. Efficient tools and techniques can reduce turnaround time, allow more comprehensive investigation in the same amount of time, or bring a target that was once too resource demanding to investigate at all into the realm of the possible.

1.2.4 Coherent

We say that an investigation is coherent if the evidence resulting from the analysis can be used to compile an integrated view of the events under question. After all of the above is said and done -

reliable, comprehensive, efficient analysis has been performed - an investigation still must use all of the evidence discovered and assemble an integrated picture of the events under question. This involves taking evidence garnered from the multitude of different tools and techniques used and forming a conclusion about answers to the questions in the investigation.

1.3 Current Practice

The current state of digital forensic investigation is plagued with problems when viewed in terms of effectiveness. First, its reliability is suspect. Current standards of practice dictate that no investigation take place using the original media (with the exception of live forensics techniques discussed in Chapter 6). Instead, a bitwise copy of the original media is made and the investigation is performed on the copy. To ensure that the copy is initially accurate, and is not tampered with during the course of the investigation, a cryptographic hash is taken of the original and the copy when it is created. Matching hashes imply that the copy is in fact accurate. At any point during or after the investigation, another hash of the copy can be computed, and checked to see that it still matches the hash of the original, and has therefore not been tampered with. This system works well enough for tamper resistance, but there are other verification issues which are not dealt with. First, there is no generally accepted method for tracking which operations were performed on the media, by whom, and when. Some investigative tools produce logs of their activity, but most do not. It is up to the investigator to track these activities manually, which is an error prone process. Even more problematic, new digital forensic tools are created all the time to address new types of targets in the field. These tools invariably have bugs, or logical errors which have the potential to invisibly compromise the integrity of the generated evidence. There is no systematic method in widespread use for determining if each of these tools in fact does what it advertises, nor for a "chain of actions" recording which operations have been performed on the media, by whom, and when.

Performing comprehensive investigations is problematic as well. There are myriad types of equipment and data which might require analysis, and because digital artifacts are not plainly visible, each requires analysis tools and techniques. While many of the tools and techniques required for these analyses are available to an investigator, many have simply not been created yet. This situation, however, has not stopped new technologies from appearing at a rapid pace, making the need for new tools even greater. Related to the challenges of performing analyses on multiple types of targets is another issue of scope, that of target size. Most tools that are currently available struggle to handle even moderately sized targets. Larger targets (on the order of terabytes) are simply beyond their ability to process. While this is clearly a comprehensiveness issue, it is also intimately related to the efficiency characteristic discussed in the next section.

Current digital forensic tools are painfully inefficient, most being single purpose tools hurriedly created to address some small piece of the puzzle with no regard for efficiency. Almost none takes advantage of multicore processors or high performance coprocessors, or makes efficient use of available IO bandwidth or storage. Exacerbating this problem is the rapid increases in the size of hard drives (and therefore the size of investigative targets) in production. This trend limits the number of targets which can be analyzed, and also limits the individual targets that can be analyzed, as some targets simply require too many resources for inefficient tools to manage. In digital forensics investigations, where the analysis targets may possess huge amount of storage, the need for IO efficiency is not to be understated. Efficiency is an important characteristic for two interrelated reasons: if individual techniques require fewer resources to apply, then a larger number of techniques can be brought to bear on the investigation in a fixed amount of time, and targets which were too large to investigate otherwise can now be analyzed.

Creation of a coherent picture of the events in question in an investigation is also something for which current techniques have difficulty. In the course of a comprehensive investigation, dozens of digital forensic tools and techniques will have been brought to bear on the target. Each provides information on some small facet of the system at different logical levels of the system hierarchy, and all of which use different output formats for results and even use different terminology for the same entities. For example, an investigation may turn up information on the processes running on the system at some point in the past, a list of recently deleted files, and a record of users' logins. Are these events related to the question under investigation? Are they related to one another? Certainly all of these describe entities on the systems which are interrelated: users, files, processes, but the job of putting together a clear picture of the relations and how to answer the questions is, as of now, a manual operation. This uses resources just as computation does, but the resources here are manpower and the cognitive powers of the investigators.

Addressing these important issues is the focus of this research. Specifically, we present several techniques for improving the efficacy of digital forensic investigations, in terms of improving their reliability, comprehensiveness, efficiency and coherence.

1.4 Contributions

This dissertation research contributes the following to the field of digital forensic investigation:

- Definition of the concept of an effective digital forensic investigation and discussion of the four characteristics thereof.
- A technique for enhancing the reliability characteristic of digital forensic investigations. We present FDAM, a module for introducing auditing to digital forensic tools.
- A technique for making digital forensic investigations (DFI) more comprehensive. We present Ramparser, a tool for the deep analysis of images of physical RAM from Linux machines.
- A set of techniques for enhancing the efficiency of digital forensics tools. We apply this set of techniques to the file carving tool Scalpel and show order-of-magnitude improvements to its performance in terms of processing speed and disk usage.
- A technique for aiding in the creation of a coherent picture of the state of a digital system. We present FACE, a tool which automates the task of integrating the data produced by a set of digital forensics tools.
- Open source tools, which will be released to the community at large to aid further research.
- Several peer-reviewed research publications have resulted from this work (see Section 1.6).

1.5 Organization

The organization of the rest of this work is as follows. Chapter 2 will give some background on the digital forensic investigative process. Each of the successive 4 chapters discusses one of the four characteristics of effective DFI, and each provides techniques for enhancing that characteristic. Chapter 3 addresses reliability, Chapter 4 focuses on comprehensiveness, efficiency is tackled in Chapter 5 and Chapter 6 discusses coherence. Finally, Chapter 7 provides conclusions and some directions for future work.

1.6 Bibliographic Attributions

Most of the material presented in this dissertation appears in previously published works.

- The material in Chapter 3 appeared in the Journal of Digital Investigation [2] in 2007 as joint work with V. Roussev and G. Richard III. The work introduced auditing techniques in DFI, and FDAM, the forensics discovery auditing module.
- The material in Chapters 4 and 6 appeared in the Proceedings of the 8th Annual Digital Forensics Research Workshop (DFRWS) [3] in 2008 as joint work with A. Case, A. Cristina, G. Richard III, and V. Roussev. This work introduced a live forensics technique for analyzing the contents of physical RAM, as well as a technique for integrating the data harvested in an investigation into a coherent whole.
- The material in Chapter 5 appeared in:
    - Proceedings of the Third Annual IFIP WG 11.9 International Conference on Digital Forensics [4] in 2007 as joint work with G. Richard III and V. Roussev.
    - Proceedings of the 7th Annual Digital Forensics Research Workshop (DFRWS) [5] in 2007, as joint work with G. Richard III and V. Roussev.
    - Handbook of Research on Computational Forensics, Digital Crime and Investigation: Methods and Solutions [6] in 2009 as joint work with S. Movva, G.

G. Richard III, V. Roussev, and L. Schwiebert. These works introduced a series of performance enhancements for digital forensics tools.

Chapter 2:
Digital Forensic Investigation Background

This section gives an overview of the digital forensics investigative process, which progresses through a set of four phases: collection, examination, analysis, and reporting (see Figure 2.1 below). In the following sections we discuss each of these phases in turn.

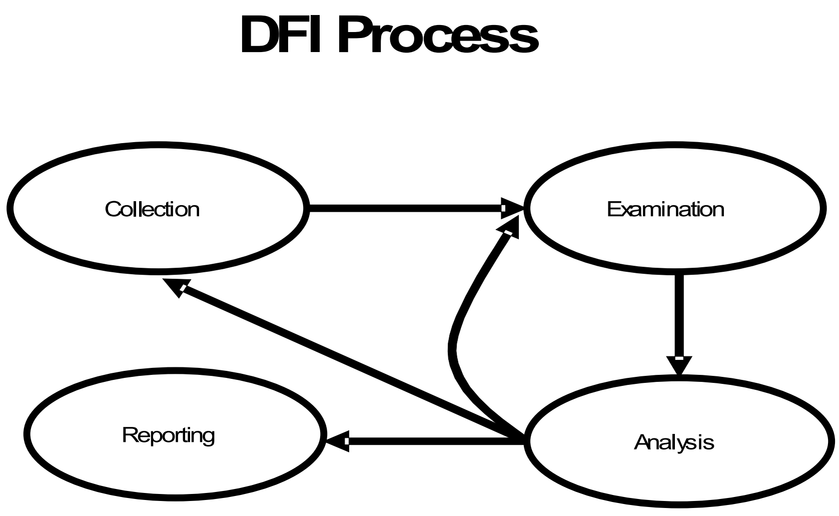


Figure 2.1: The Digital Forensic Investigative Process.

2.1 Collection

The initial phase, evidence collection, entails acquiring bitwise copies, termed *images*, of all media which are suspected of containing evidence. The types of media include, but are not limited to: hard drives, USB devices, physical RAM, CDs / DVDs and SD cards. Great care must be taken to ensure that not only are the copies accurate, as they will be the basis of the subsequent investigation, but also that the original media is not altered in the process. Any alteration of either the copy or the original can render the entire investigation invalid. In order to support the integrity of this process several tools have been created, some implemented in hardware and others in software. On the hardware side, write blockers such as [7] [8] allow the investigator to attach a drive to a special purpose controller which will disallow any write operation from taking place on the connected drive. This allows for the ability to create an image of a drive with no fear of altering the original. General purpose CD and DVD duplicators are used to make copies of those specific types of media. PC expansion cards like Tribble [9] can create images of volatile RAM and write the copy to external media. On the software side there are write blockers which perform similarly to their hardware counterparts [10], as well as several other tools for acquiring images, like FTK Imager [11] and dcfldd [12]. Tools such as mount [13] on Linux systems have facilities for accessing a disk to make a copy while ensuring that the disk is not written to. Once an image is acquired, its integrity is verified, generally by generating a cryptographic hash like MD5 [14] or one of the SHA [15] family, of both the original and the copy, and then comparing the two. The properties of the hashes used ensure that matching hash values imply that the underlying data is in fact the same and the copy is accurate. Last, the original media is securely stored away, and the remainder of the investigation utilizes the images. Note that the storage requirements for this phase are on the order of the size of the original media, which in some cases can be quite large (terabytes or greater).

2.2 Examination

The next phase, examination, entails enumerating the objects on the media. We use the term *objects* here to mean any entity of forensic interest; as this changes for each investigation and for investigations in general over time, we discuss only a small subset of the possibilities. This enumeration occurs across multiple logical levels of a system, operating on multiple object types, some examples of which are discussed here. For example with hard drives, generally during this phase, partitions, operating systems, filesystems, and files themselves are catalogued using tools like The Sleuthkit [16]. Files are categorized as one of numerous types, including documents, images, logs, or configuration files. Deleted files are recovered (where possible) using file carvers like Scalpel [17]. Several types of preprocessing can occur in this phase, including thumbnail generation for digital photographic image files, hashing of individual files for filtering and duplicate detection. The National Software Reference Library (NSRL) [18] contains a list of hashes of known-benign files, such as standard operating system or application files. Using this list, and the hashes of the files on the system under investigation, we can filter out of an investigation those files which are of no forensic interest. Similarly, we can use file hashes to quickly detect duplicate files on the system. A full text index can be generated, using DTSearch [19], or using the Lucene [20] API over the raw drive, in order to facilitate later keyword searches in the next phase. A timeline of filesystem activity can be generated using mactime [21]. All files on the media can be scanned for evidence of malware using any number of scanners, including Norton [22] and McAfee [23]. Items that may be of particular interest, like deleted files, or files with incorrect extensions can be flagged as such. Encrypted and password protected files can be singled out and password recovery tools such as Cain & Abel [24], or Access Data's Password Recovery Toolkit (PRTK) [25] can be brought to bear. Important to note here is that this phase is tool-centric in that the tools do the majority of the work. This point differentiates the current phase from the next, analysis, during which the burden is shifted to the investigator. Many of the functions listed here are performed in the majority of investigations as a sort of baseline, and several all-in-one suites, such as FTK [26], encase [27] and PyFlag [28] exist to automate the process, and collect all of the results under one larger tool.

2.3 Analysis

During the analysis phase, the data generated in the previous phase is used to begin to answer the questions under investigation. Where examination is mostly an enumeration of the objects in the system, analysis attempts to assign meaning to the objects in the context of the investigation. Where we may have acquired and categorized files as email before, we now analyze (here, read) them in order to determine if they are evidence which can be used to prove or disprove our assertions. Much of the work here is cognitive on the part of the investigator, given minimal support from tools which allow him to view the objects in the system. Evidence uncovered as part of the analysis phase might require a jump back to the collection phase, in order to collect other media, or to the examination phase, to enumerate types of objects on the forensic target which were not previously examined.

2.4 Reporting

The final phase of the digital forensic investigative process is report generation. In this phase any conclusions drawn from the analysis phase are compiled into a report. The report generally contains conclusions, and the methodology used in the investigation which led to said conclusions, and information required to verify the integrity of the investigation, such as a list of the evidence media with their respective hash values, and forms verifying the chain of custody for the evidence media. While some of the tool suites discussed above do have automatic report generation features, most single purpose tools do not. It is left to the investigator to record his methodology and manually construct a full report.

With this high-level understanding of the digital forensic investigative process, we now move on to a more thorough discussion of each the characteristics of an effective digital forensic investigation. In each of the following 4 chapters we address one of theses characteristics in terms of the current state of practice, deficiencies therein, and techniques which foster more effective investigations. We begin with reliability.

Chapter 3:
Reliability

Given that digital forensic investigations (DFIs) have been the basis for sentencing people to prison, as well as proving their innocence in the face of criminal charges, the need for reliability is clearly important. For the purposes of this research, we use the term reliability not in the statistical sense as a measure of the consistency of a repeated set of measurements, but as a measure of the trustworthiness of the results of an investigation in the face of potential for errors on the part of the investigator or the tools used. If the results of an investigation are not trustworthy, then the entire investigative concept is undermined. In current practice, ensuring that DFIs are reliable is mostly an ad-hoc process manually performed by the investigator. In this chapter we detail how reliability issues are dealt with currently, point out failures in the system, and provide a technique for improving reliability in DFI.

3.1 Current Practice

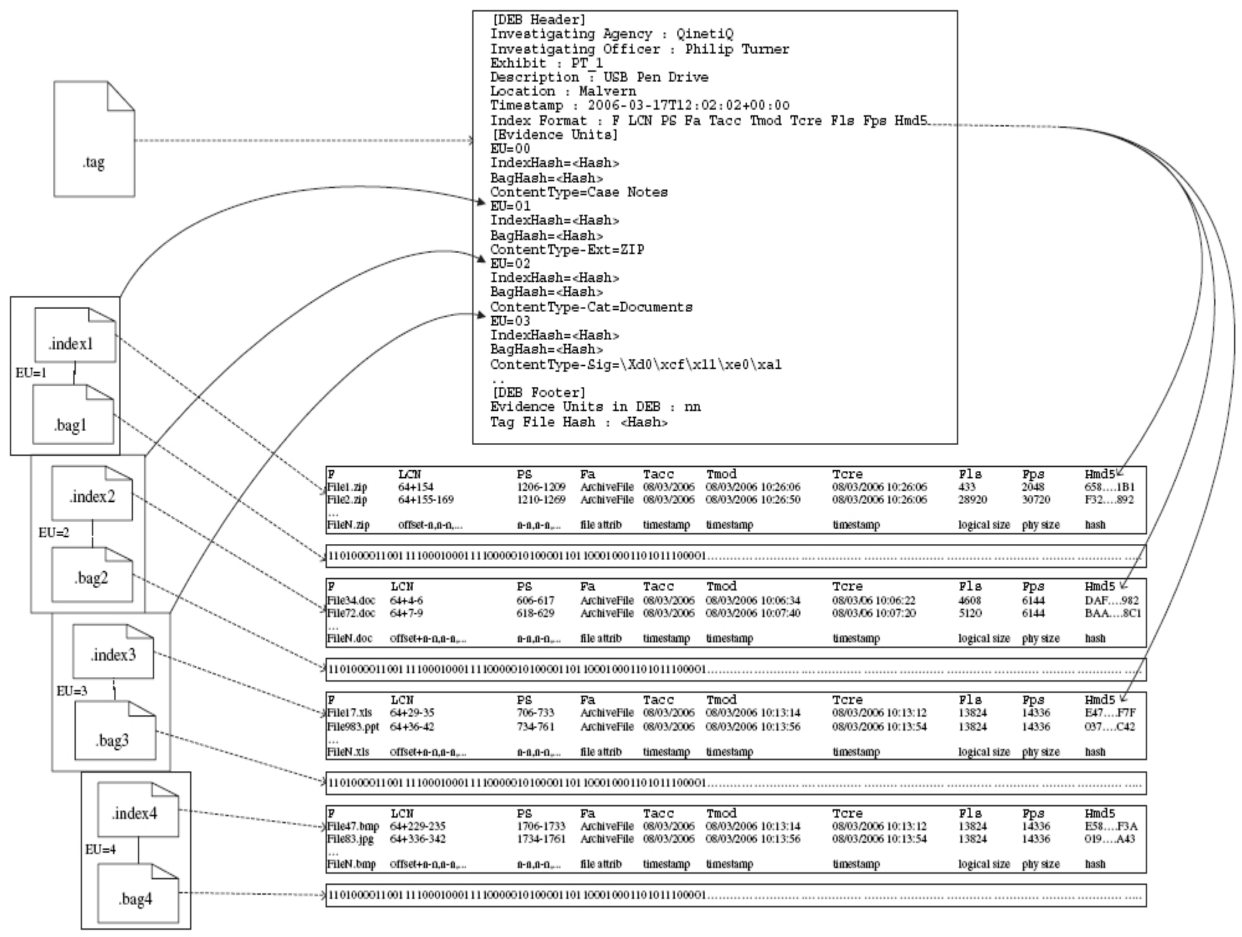
In a standard physical investigation, reliability begins with the collection of evidence at the crime scene. The collected evidence is managed by storing it in physical evidence bags which aid in the performance of several functions. They preserve the state of the stored object, and provide evidence of tampering if it occurs. Chain of custody of the evidence is preserved by maintaining a list of the persons who have accessed the evidence on the bag itself. Other metadata about the case, such as case number, evidence item number, time and place of collection are recorded there as well. Then the evidence is analyzed by the detectives with support from their forensics team and in the end a report is generated detailing the findings of the investigation.

3.1.1 Collection

DFIs deal with digital artifacts, so collection begins with acquiring an image of some digital device. Of primary concern here is that the copy is accurate, and that the original media is not altered. This process can be performed using hardware or software acquisition tools. In order to ensure that the original media is not altered, acquisition tools use write-blocking techniques. Hardware tools intercept accesses to the original media and do not allow write operation to occur, while allowing read operations to pass unfettered. Software techniques work similarly either by disallowing writes themselves, or by making sure to mount the original media as read-only at the block device or filesystem level. In order to guarantee that the copy is accurate, cryptographic hash signatures like MD5, or those of the SHA family, are generated for the original media and the copy. Matching hashes imply that the original and copy are in fact the same.

3.1.2 Preservation

In DFI, where there is no physical container like the evidence bag in which to store objects, digital evidence is stored in files on an investigation machine. This begs the question: how are the functions performed by physical evidence bags duplicated in a DFI? In the simplest case, these functions are performed in an ah-hoc manner by the investigator using a host of tools. Tamper evidence is achieved by relying on the hash signatures generated on collection. At any time the hash of the copy can be re-generated such that if any bits of the media have been changed, the successive hashes generated will differ from the original hash. Chain of custody is preserved by keeping records of accesses to the original media, as well as the copy. Other metadata about the object is also recorded and stored by the investigator on some media, paper or electronic. This process is ad-hoc, subject to omission and error prone at best. In order to address this, the community is moving toward the use of digital evidence containers (DECs).



```
[DEB Header]
Investigating Agency : QinetiQ
Investigating Officer : Philip Turner
Exhibit : PT_1
Description : USB Pen Drive
Location : Malvern
Timestamp : 2006-03-17T12:02:02+00:00
Index Format : F LCN PS Fa Tacc Tmod Tcre Fls Fps Hmd5
[Evidence Units]
EU=00
IndexHash=<Hash>
BagHash=<Hash>
ContentType=Case Notes
EU=01
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Ext=ZIP
EU=02
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Cat=Documents
EU=03
IndexHash=<Hash>
BagHash=<Hash>
ContentType-Sig=\Xd0\xcf\x11\xe0\xa1
..
[DEB Footer]
Evidence Units in DEB : nn
Tag File Hash : <Hash>
```

F	LCN	PS	Fa	Tacc	Tmod	Tcre	Fls	Fps	Hmd5
File1.zip	64+154	1206-1209	ArchiveFile	08/03/2006	08/03/2006 10:26:06	08/03/2006 10:26:06	433	2048	658….1B1
File2.zip	64+155-169	1210-1269	ArchiveFile	08/03/2006	08/03/2006 10:26:50	08/03/2006 10:26:06	28920	30720	F32….892
…									
FileN.zip	offset-n,n-n,…	n-n,n-n,…	file attrib	timestamp	timestamp	timestamp	logical size	phy size	hash

```
1101000011001 1110001000111110000010100001101 1000100011010111000001……
```

F	LCN	PS	Fa	Tacc	Tmod	Tcre	Fls	Fps	Hmd5
File34.doc	64+4-6	606-617	ArchiveFile	08/03/2006	08/03/2006 10:06:34	08/03/06 10:06:22	4608	6144	DAF….982
File72.doc	64+7-9	618-629	ArchiveFile	08/03/2006	08/03/2006 10:07:40	08/03/06 10:07:20	5120	6144	BAA….8C1
…									
FileN.doc	offset+n-n,n-n,…	n-n,n-n,…	file attrib	timestamp	timestamp	timestamp	logical size	phy size	hash

```
1101000011001 1110001000111110000010100001101 1000100011010111000001……
```

F	LCN	PS	Fa	Tacc	Tmod	Tcre	Fls	Fps	Hmd5
File17.xls	64+29-35	706-733	ArchiveFile	08/03/2006	08/03/2006 10:13:14	08/03/2006 10:13:12	13824	14336	E47….F7F
File983.ppt	64+36-42	734-761	ArchiveFile	08/03/2006	08/03/2006 10:13:56	08/03/2006 10:13:54	13824	14336	037….C42
…									
FileN.xls	offset+n-n,n-n,…	n-n,n-n,…	file attrib	timestamp	timestamp	timestamp	logical size	phy size	hash

```
1101000011001 1110001000111110000010100001101 1000100011010111000001……
```

F	LCN	PS	Fa	Tacc	Tmod	Tcre	Fls	Fps	Hmd5
File47.bmp	64+229-235	1706-1733	ArchiveFile	08/03/2006	08/03/2006 10:13:14	08/03/2006 10:13:12	13824	14336	E58….F3A
File83.jpg	64+336-342	1734-1761	ArchiveFile	08/03/2006	08/03/2006 10:13:56	08/03/2006 10:13:54	13824	14336	019….A43
…									
FileN.bmp	offset+n-n,n-n,…	n-n,n-n,…	file attrib	timestamp	timestamp	timestamp	logical size	phy size	hash

```
1101000011001 1110001000111110000010100001101 1000100011010111000001……
```

Figure 3.1: Digital Evidence Bag [30].

DECs are containers constructed using normal operating system files, which, in an attempt to more closely duplicate the functionality of physical evidence bags, bundle evidence objects with some set of associated metadata. Below, we discuss some of the DECs currently in use.

Turner's Digital Evidence Bags (DEBs) [29] [30] [31] (Figure 3.1) store evidence objects and associated metadata in a file hierarchy. At the top is the tags file, which contains high-level metadata such as a DEB identification number, the time and date the evidence was captured and by whom, and a list of the Evidence Units (EUs) contained in the DEB. Also in the tags file are Tag Continuity Blocks for tracking operations performed on the DEB, and a description of the format of the index files. The index files contain listings of the contents of their associated bag files. Bag files contain actual evidence objects. Evidence objects can be binary blobs of data or sets of files.

The Advanced Forensics Format (AFF) [32] [33] and its newer redesign, AFF4 [34], are designed specifically to be open and extensible. They are free, open source, and free from overly restrictive intellectual property constraints. AFF supports the definition of arbitrary metadata by storing all data as name and value pairs, called segments. Some segments store the disk data and others store metadata. Because of this general design, any metadata can be defined by simply creating a new name and value pair. Each of the segments can be compressed to reduce the size of drive images, and cryptographic hashes can be calculated for each segment to ensure data integrity. AFF supports two compression algorithms: zlib, which is fast and reasonably efficient, and LZMA, which is slower but dramatically more efficient. New versions of AFF also support encryption of disk images. Also provided is the afflib [35] and a set of tools which allow for easy creation and manipulation of AFF containers.

Generic Forensic Zip (Gfzip) [36] is similar to AFF, to the point of having a reasonable level of interoperability. It provides facilities for compression, and uses strong x.509 certificates for signing data and metadata.

Seekable Gzip (Sgzip) [37], used by PyFlag, is a version of gzip compression which allows fast seeks through a file without having to uncompress it in its entirety. It achieves this by compressing the given file in separate chunks (32k by default) such that only the needed data chunk need be uncompressed. It does not, by itself, associate any metadata with image files compressed with it.

The de facto standard for DECs is the proprietary Encase file format [38]. Based on the expert witness compression format (EWF), it is the default container for the Encase and the FTK forensics tool suites and the LinEn [39] Linux disk acquisition tool. Images can be stored compressed or not, segmented or in a single chunk. Metadata is bundled in the container, the header for which contains the date and time of acquisition, an examiner's name, notes on the acquisition, and an optional password. CRCs are recorded throughout in order to detect damage to the data. Several other formats are in use (dcfldd, DIBS USA Rapid Action Imaging Device (RAID) [40], and ProDiscover Image File Format [41] to name a few), but most have functionality similar to those listed above.

Of note also is the fundamental "container" on which other are based, namely the dd raw format, which is simply a byte for byte copy from the source disk to a standard operating system file, with no associated metadata besides that recorded by the filesystem.

Using DECs to replicate some of the functionality of physical evidence bags resolves some of the issues faced in performing reliable DFIs, but there are many other important issues which they do not address.

3.1.3 Correctness

Because digital artifacts are not visible to the naked eye, investigation requires tools to make the data visible. As technologies rapidly change, new tools must be developed for analysis of the new data types produced by these new technologies. In the face of rapid development, we require methods for guaranteeing that these tools perform as advertised. This is difficult for two reasons. First there is simply a large number of tools, too many on which to perform exhaustive correctness testing. Second, the datasets the tools operate on, and the resulting data they generate, are also extremely large. This second point is due in part to one of the main differences between physical and digital investigations – in the physical investigation of a crime scene, investigators pick and choose items to collect as evidence for later analysis, whereas is an digital investigation, the entire crime scene (digital device) can be collected or perfectly copied. This leads to a glut of data which must be analyzed. Verifying nearly any level of correctness manually is doomed to failure. There are two techniques in current use to mitigate these problems.

The NIST [42] Computer Forensic Tool Testing Project (CFTT) [43] seeks to "establish a methodology for testing computer forensic software tools by development of general tool specifications, test procedures, test criteria, test sets, and test hardware." The CFFT produces unbiased reports on the correct performance of commonly used tools when faced with a standard set of test inputs. This can provide guidance to investigators as to which tools are most likely to give accurate results, and make them aware of common errors made by the tools they use.

In a similar vein, in [44] Simpson et al introduce the Digital Corpora [45], a collection of data sets to be used as standard forensic corpora for tool testing. These corpora will allow for correctness testing of tools, for measuring relative performance, and other issues related to the production and use of reliable DFI tools and techniques. Several other less formalized data sets are available as well; see [46] for a listing.

3.1.4 Audit Trails

A normal DFI might involve the use of a large number of special purpose tools executed over the course of the investigation with different tool used each time. In order for the results of the investigation to be trustworthy, a detailed report must be generated, cataloguing the methodology used to reach the investigation's conclusions. We refer to the process of recording all operations on the evidence as auditing. In order to support this, some tools produce audit logs of the operations performed using them. Those that do support audit trails include mostly large tool suites, such as FTK and Encase.

3.2 Issues

Write blockers, cryptographic hashes, DEBs, tool testing, and other techniques help in making DFIs more reliable, but there are still several weaknesses that must be addressed in the tool correctness and tool auditing areas to support reliable investigations.

On the correctness side, it is a simple fact that software and hardware forensics (and other) tools have bugs. Some image acquisition tools mishandle errors on the original media, either skipping bytes near the error, writing blocks with errors as zeros or just silently skipping blocks of input when errors are encountered. Some tools fail to acquire the last sector of disks for disks having an odd number of sectors [145]. Yet others fail to copy special disk areas, like the Host Protected Area (HPA). See [47] [48] [49] for examples. That these types of errors exist for tools whose functionality is so easily described ("copy every byte accurately and report errors") should be a clear indicator of the need for some type of correctness verification for these and more complex tools, such as those which parse complicated file formats, or create highly optimized full-text indexes. The NIST CFTT provides a usable baseline, but it has at least one major drawback – the testing done provides only a snapshot-in-time measure of correct performance. Newer versions of the tested tools can introduce new bugs and therefore require re-testing. Additionally, the logic of the underlying data beneath the tools can change, rendering their previously correct operation incorrect. If, for example, the file format specification for some common document type is changed, the tools which parse these documents must be updated to reflect the changes. In this situation, continuing to use a known-good version of the parser document file parser can lead to incorrect results. The NIST technique also requires substantial testing for each tool to verify its correctness; this is a daunting challenge given the number of tool in use and the rate of production of new tools.

Even if the correctness problem were solved, there are also reliability issues with regard to auditing. The object of an investigation is to produce evidence supporting or refuting some conjecture. If this evidence is to be relied on, the methodology used to generate that evidence must be available for verification – without reproducibility the results of any investigation would be suspect. The task of recording which operations were performed on the data collected during an investigation is left largely to the investigator to perform manually. The sheer number of tools and techniques which may have to be used makes this difficult. Some tools generate audit logs of their activities, but most do not. Even those that do audit their activities produce different amounts of logging information, and in formats generally incompatible with the outputs of other tools. Moreover, are we to trust a tool to perform its own auditing in light of the correctness issues discussed above? Even if we do, the investigator must still perform copious amounts of manual bookkeeping, or be forced to use only one tool or tool suite which generates a usable, correct audit log. Some DECs do provide hooks which allow generic tools to produce audit logs, but these require voluntary participation on the part of the tools, and instrumentation of the tools to use the presented APIs. In order to mitigate these problems of correctness and auditing, we present the Forensic Discovery Auditing Module (FDAM).

3.3 FDAM

FDAM provides two types of functionality: it enforces restrictions on attempts to access evidence, and monitors the accesses it allows. Several design goals aid in these processes.

- *Audit Logging*. The main purpose of the FDAM is to provide a complete and trustworthy history of all data operations performed on the forensic image. Logging should be performed both at the filesystem layer and at the block-device layer to fully document all operations. Ideally, it should support various levels of detail (based on user needs) and most common formats currently in use.
- *Flexible DEC Support*. It should be able to import from and export to multiple digital evidence container formats. This includes both specialized forensic containers, such as DEB and AFF, and generic ones such as plain files and raw device images.
- *Tool Independence*. The module should work for any application used in the forensic process, including 'non-forensic' ones.
- *Tool/Agent Identification*. The module should unambiguously identify the tool used to perform each operation (along with its run-time parameters) and should be able to attribute it to the user on whose behalf the operation is executed.
- *Policy Enforcement*. Once it is possible to identify and attribute tool use, the next step is to enable policy enforcement by blocking undesirable behavior. Policies can be based on the type of tool used (either black-list, or white-list), user identity (access control), or tool behavior (e.g. no write operations, no block-level operations). The last option allows for the safe use of general-purpose tools that have not been tested (or certified) for forensic use.

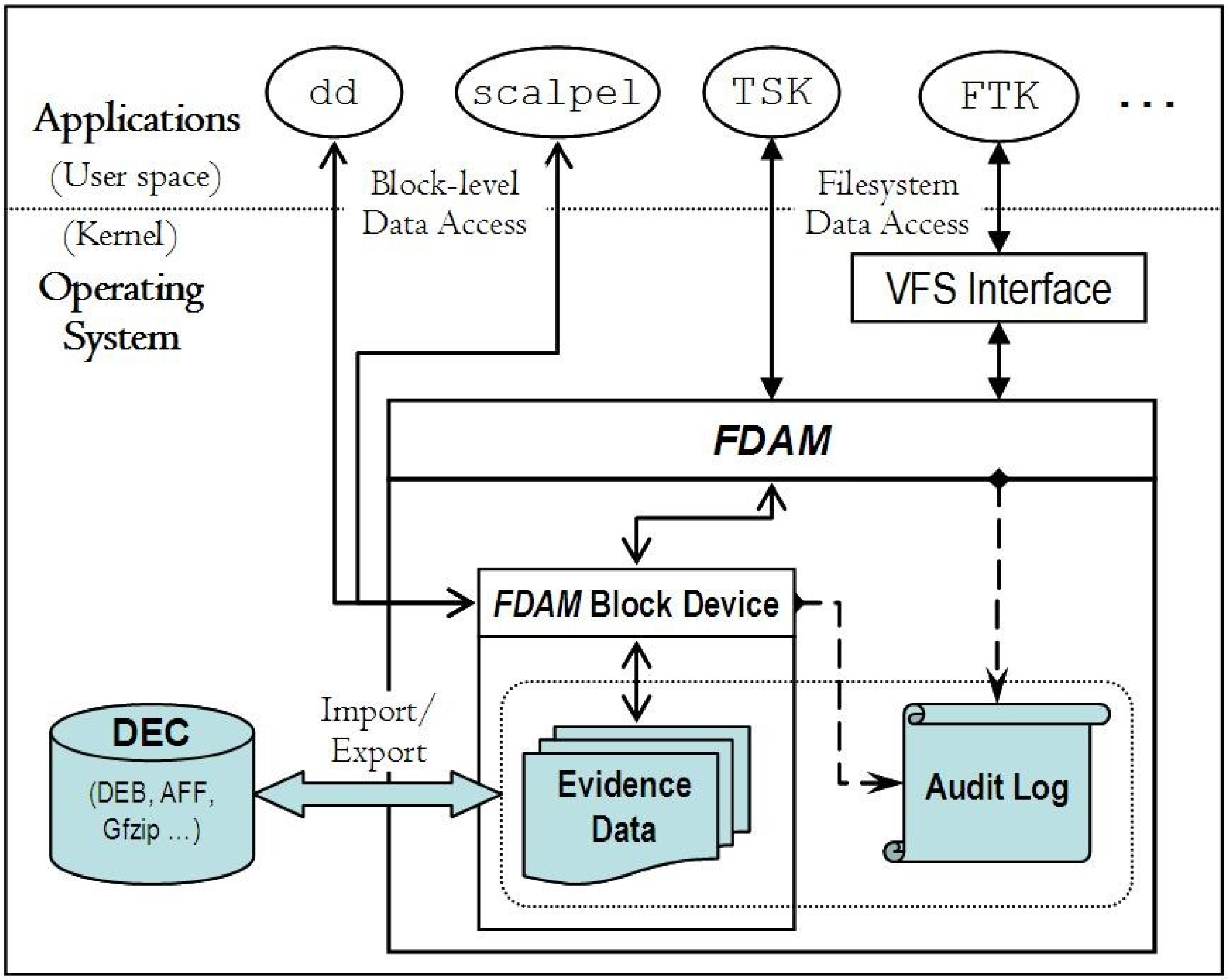


Figure 3.2: FDAM Architecture.

3.3.1 Architecture

From a system design perspective, the logical place to install independent auditing is the operating system, specifically, the filesystem interface. Since most tools rely on operating system components to interpret the file system found on the forensic image (as opposed to interpreting the raw image), installing an auditor here is a logical choice. We note that, for completeness, the auditor should also be installed at the lower, block-level interface to document operations that bypass the filesystem interface. Such a discussion is beyond the scope of this work, however in [4], we have demonstrated the use of a similar block-level device. The only addition it needs is the secure logging, which would be substantially the same as with filesystem operations. The

FDAM architecture is shown in Figure 3.2 and operates as follows. First, DEC containers (in various supported formats) are imported by the kernel module, which results in the creation of a block-level device (raw image) of the evidence data, and a mounted filesystem (if possible), which are then presented to the applications for processing, as usual. We should point out that the import does not necessarily mean that a new physical raw image is actually created. In fact, we expect that the typical behavior would be to wrap the existing DEC with block-device/filesystem layer interfaces. Once the DEC is imported, all access is monitored and logged. Finally, upon completion of the examination, the raw data and the logs are packed and sealed in a DEC of choice. We expect that different types of DECs will be used, depending on the size and complexity of the case. Under our scheme, it becomes possible to perform and document conversions between different formats, as well as splitting and aggregation. Since all operations are automatically recorded in a trusted log, it is straightforward to demonstrate chain of custody and to verify integrity in an automated manner. The following sections survey implementation choices for our design, discuss how native applications can transparently access DECs, and describe some actual experience with our prototype implementation.

3.3.2 Implementation Choices

We evaluated the capabilities of several filesystems before choosing a candidate for native DEC support, including ext2/3, ReiserFS, NTFS, and FUSE (File System in Userspace) [50]. NTFS was eliminated from consideration because source code is not available, although NTFS alternate data streams are an attractive mechanism for implementing DEC resource forks (e.g., blobs of digital evidence, the audit log, and DEC metadata). Most of the other filesystems we considered contain similar features that might be used to support efficient manipulation of DECs, such as support for extended attributes (EAs). But the EA support in several of the candidate filesystems is limited. FUSE offers reasonable performance, flexibility and verifiability. New ideas are straightforward to implement and only a relatively small amount of additional code needs to be verified for forensic soundness. Further, the user-level FUSE code is segregated from kernel-level filesystem code. We briefly discuss FUSE before providing some technical details about our implementation.

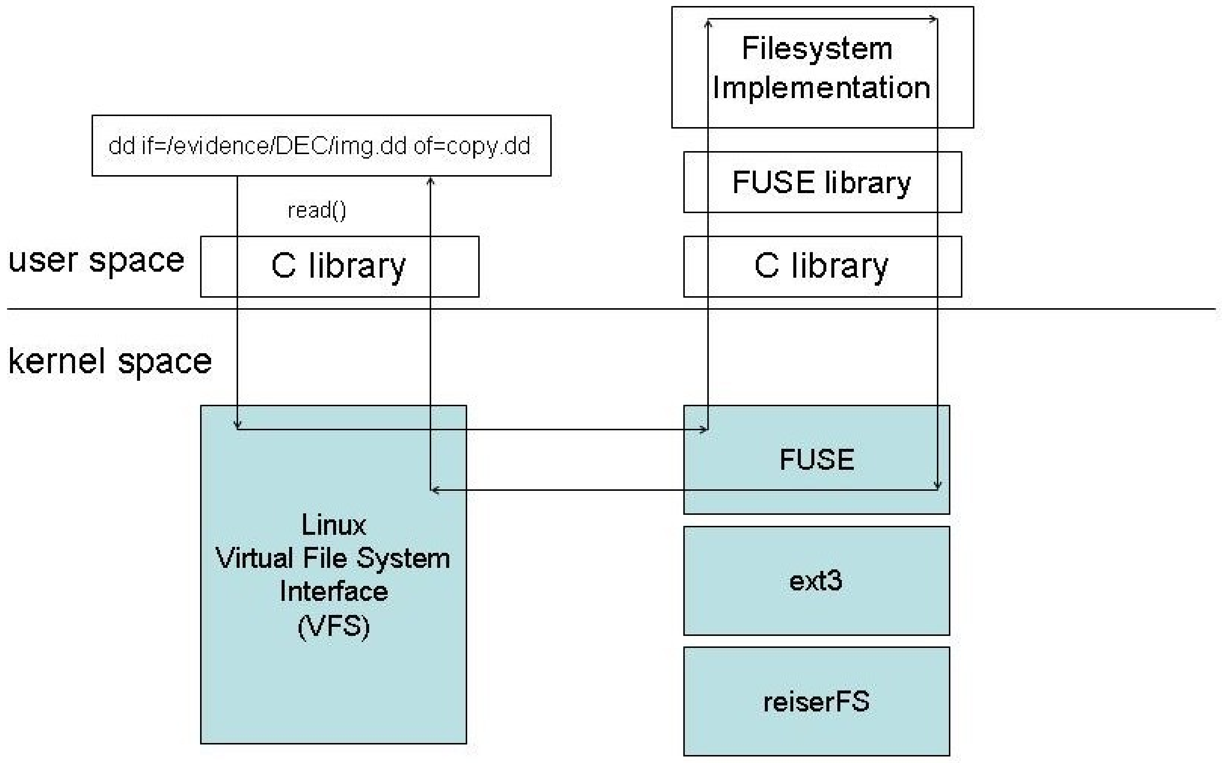

Filesystem Implementation
dd if=/evidence/DEC/img.dd of=copy.dd
FUSE library
read()
user space
C library
C library
kernel space
FUSE
Linux Virtual File System Interface (VFS)
ext3
reiserFS

Figure 3.3 FUSE: Architecture.

3.3.3 FUSE Filesystem in Userspace

FUSE is a system for rapid development of filesystems. The architecture of FUSE is illustrated in Figure 3.3. A FUSE kernel component, which implements a Virtual File System (VFS) filesystem, traps system calls and redirects them to a userspace filesystem implementation, which is compiled against the FUSE library. This allows new filesystems to be quickly designed and built without the complexity of in-kernel hacking. The FUSE kernel module acts as a bridge to the VFS kernel interfaces. To instrument system calls in FUSE, the new filesystem supplies a structure *fuse_operations* that redirects system calls through functions defined by the filesystem. This structure is depicted in Figure 3.4 below.

```
static struct fuse_operations aud_oper = {
   .getattr = aud_getattr,
   .access = aud_access,
   .readlink = aud_readlink,
   .readdir = aud_readdir,
   .mknod = aud_mknod,
   .mkdir = aud_mkdir,
   .symlink = aud_symlink,
   .unlink = aud_unlink,
   .rmdir = aud_rmdir,
   .rename = aud_rename,
   .link = aud_link,
   .chmod = aud_chmod,
   .chown = aud_chown,
   .truncate = aud_truncate,
   .utime = aud_utime,
   .open = aud_open,
   .read = aud_read,
   .write = aud_write,
   .statfs = aud_statfs,
   .release = aud_release,
   .fsync = aud_fsync,
   #ifdef HAVE_SETXATTR
   .setxattr = aud_setxattr,
   .getxattr = aud_getxattr,
   .listxattr = aud_listxattr,
   .removexattr= aud_removexattr,
   #endif
}
```

Figure 3.4: struct *fuse_operations*.

Each of these functions can completely redefine a filesystem operation, augment its functionality, or provide auditing. For example, to audit write operations, a filesystem can define the function *aud_write()* to record information about the calling process and then call the standard FUSE function to carry out the write operation. The function *fuse_get_context()* allows the UID, GID, and process ID of the calling process to be obtained within each file operation. This information can then be supplemented with any of the plethora of information in the /proc pseudo filesystem (under Linux) to build the context for the write, such as the name of the application, or the exact command line invocation.

FUSE currently has ports for Linux and FreeBSD and a number of language bindings, including C, C++, Python, and Perl. FUSE is being actively developed and is in widespread use; one of the most important uses of FUSE is for NTFS support in Linux, via ntfs-3g [134]. As of kernel version 2.6.14, FUSE is integrated into the Linux kernel.

3.3.4 Prototype

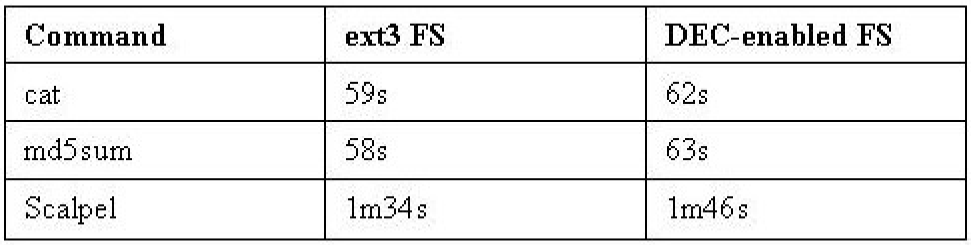
We have developed a prototype system for native filesystem support for both DEC-enabled and legacy applications, based on FUSE. Our system is developed in C and Python, under Linux. In our prototype, user-level applications are currently used to import and export DECs into and out of a special DEC-aware FUSE filesystem. An import operation essentially splits the DEC into component files and places these files into the filesystem, along with the DEC audit log and other metadata. Exporting a DEC from the DEC-enabled filesystem simply recreates the DEC structure from the data stored in the corresponding directories in the filesystem. The use of these import and export applications enables our system to be neutral with regard to developing standards for DEC structure. Our prototype provides automatic auditing of access to DECs by applications. Many applications, including those specifically designed for digital forensics investigation (e.g., file carvers) and those which are not (e.g., *dd* and other common Unix command line programs) may never be modified for DEC compliance. So our implementation instruments filesystem-level system calls (through FUSE), such as file open, read, and write operations, and captures information about both the calling application and the operations themselves. Applications may simply use the standard C library *open()*, *close()*, *read()*, and *write()* operations (and their buffered counterparts) on digital evidence blobs contained within a DEC. Access to the blobs of digital evidence in a DEC automatically results in updates of the audit log. For example, an open operation records information including the user ID, process ID, MD5 hash of the accessing application's executable, the date and time, and the command line of the accessing application. Auditing of read/write operations ties these operations to the associated open operation and writes entries in the log detailing the offsets and lengths of the data accessed. To illustrate, fragments of an audit log for a DEC containing a single disk image are illustrated in Figure 3.5. The contents of the DEC were accessed by a number of applications, including tools from the Sleuthkit and a file carver. The actual log is much larger; these fragments are provided to illustrate the information gathered on each operation.

```
Thu Aug 31 18:07:43 2006 uid: 0, gid: 0, pid: 22071
eid: 1 open: /root/work/audit_fs/evidence/ext2_image/ext2_image
```

```
executable: /usr/bin/fsstat
hash: 7b16b5e9abf862528f54d2405686aa27
execution: fsstat ext2_image
Thu Aug 31 18:07:43 2006 eid: 1 read length: 32768, starting at: 0
Thu Aug 31 18:07:43 2006 eid: 1 read length: 4096, starting at: 65536
Thu Aug 31 18:07:43 2006 eid: 1 read length: 4096, starting at: 262144
Thu Aug 31 18:07:43 2006 release:
/root/work/audit_fs/evidence/ext2_image/ext2_image
...
...
Thu Aug 31 18:07:51 2006 uid: 0, gid: 0, pid: 22083
eid: 2 open: /root/work/audit_fs/evidence/ext2_image/ext2_image
executable: /usr/bin/sleuthkit/ils
hash: 5b6e1d30a8b02d5d01adc24c4bc7b57b
execution: ils ext2_image
Thu Aug 31 18:07:51 2006 eid: 2 read length: 32768, starting at: 0
Thu Aug 31 18:07:51 2006 eid: 2 read length: 4096, starting at: 65536
Thu Aug 31 18:07:51 2006 eid: 2 read length: 4096, starting at: 262144
Thu Aug 31 18:07:51 2006 eid: 2 read length: 16384, starting at: 32768
Thu Aug 31 18:07:51 2006 eid: 2 read length: 16384, starting at: 49152
...
...
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 940949504
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 941080576
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 941211648
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 941342720
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 941473792
Thu Aug 31 18:07:52 2006 eid: 2 read length: 131072, starting at: 941604864
Thu Aug 31 18:07:52 2006 release:
/root/work/audit_fs/evidence/ext2_image/ext2_image
...
...
Thu Aug 31 18:08:04 2006 uid: 0, gid: 0, pid: 22097
eid: 5 open: /root/work/audit_fs/evidence/ext2_image/ext2_image
executable: /usr/local/bin/scalpel
hash: 8ab31b90d800ed36da88cdaa5176aaaa
execution: scalpel ext2_image
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 0
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 131072
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 262144
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 393216
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 524288
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 655360
Thu Aug 31 18:08:04 2006 eid: 5 read length: 131072, starting at: 786432
...
...
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072, starting at: 744095744
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072, starting at: 744226816
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072, starting at: 744357888
Thu Aug 31 18:09:32 2006 eid: 5 read length: 131072, starting at: 744488960
Thu Aug 31 18:09:32 2006 release:
/root/work/audit_fs/evidence/ext2_image/ext2_image
```

Figure 3.5: Sample FDAM Output.

To provide an acceptable balance between performance and thorough auditing, our prototype allows many options to be configured on installation. One option is automatic hashing of the executables of applications accessing a DEC (e.g., when an open operation is performed). Another is automatic hashing of components of the DEC, as well as the entire DEC, after certain operations such a *write()* or a *close()*. To conclude this discussion, we note that the amount of information that can be gathered by our prototype about the calling application and the state of the DEC is enormous—for example, in some circumstances it might be useful to track which other files a application has open, which network connections are open, the BIOS version of the machine performing the investigation, etc. Configurability allows an appropriate balance between performance and adequate oversight of an investigation.



Command	ext3 FS	DEC-enabled FS
cat	59s	62s
md5sum	58s	63s
Scalpel	1m34s	1m46s

Table 3.1: FDAM Performance Overhead.

3.3.5 Performance Study

We ran a number of experiments to determine the overhead of auditing access to DECs. These experiments target unmodified legacy applications accessing blobs of digital evidence stored in a DEC inside of our prototype DEC-enabled filesystem. For these experiments the Python implementation of the filesystem was used. The C version generally exhibits slightly better performance. Table 3.1 presents representative performance results. We ran *cat*, *md5sum*, and *Scalpel* on a 960MB ext2 disk image, stored in an ext3 partition and in our DEC-enabled filesystem. For each application, three executions were timed, with reboots of the test machine between each run to minimize caching effects. The average of the three runs is reported. The overhead for access to digital evidence by legacy applications in the DEC-enabled filesystem varies from 5-13%. Scalpel, a file carver, incurs more overhead because it makes two passes over the entire disk image. In a number of additional experiments, we've observed an average overhead of approximately 9%. Additionally, we note that accesses over network file sharing applications such as Samba obfuscate the name of the application touching a blob of digital evidence. For example, if a Windows application accesses DEB data through a Samba share, the audit log shows only *smbd* as the accessing application (i.e., the Samba daemon under Linux). Such limitations illustrate the need for additional work on support for networked access to DECs.

3.4 Discussion

In this chapter we have discussed the issue of reliability in DFI, including current practice and some of its weaknesses. Current tools and techniques do not provide a mechanism for tracking which operations have been performed on the evidence data and by whom and when. This forces the investigator to manage tracking the steps of the investigation manually, an error-prone process. To provide for more reliability in DFI we presented FDAM, a module for auditing and controlling access to digital evidence by forensics tools. FDAM provides a virtual filesystem which monitors all accesses made to files kept within it, and logs access to protected files. The level of logging is configurable, and so can record the time and date of accesses, which user ran the accessing process, and even which parts of the evidence data the tool read or wrote. Records of exactly which pieces of the evidence files were touched can be used to audit the correct operation of the tools used. In addition to monitoring, FDAM can control access, e.g. blocking all write operations to evidence files. These functions together can protect evidence, and keep records of the context of accesses, increasing the reliability of the investigation itself. In the next chapter, we change focus to another feature of effective DFI, comprehensiveness.

Chapter 4: Comprehensiveness

When we attempt to enumerate all of the interesting types of data that could be part of an investigation, we quickly see that the list is nearly endless, and yet it continues to grow. As each of these types is a digital artifact and cannot be plainly seen, each requires a tool implementation in order to facilitate effective analysis. While these tools exist for basic investigation of many of the more commonly seen data types, for many others they do not. In this chapter we discuss the need for new tools to keep up with advances in technology in order to support comprehensive investigations.

4.1 Current Practice

Digital forensic investigation (DFI) has at its disposal reasonable tool coverage for many types of data. Here we detail the coverage provided by existing tools and techniques. Note that new tool appear daily, and there is no central point of advertising or distribution, so an exhaustive study is impossible; we will however attempt to provide a reasonably accurate picture as of this writing. We proceed by organizing tools by their functionality based on the types of targets they apply to.

4.1.1 Disk Forensics

By far the most common type of device on which to perform DFI is the hard drive. It also has by far the most tools devoted to its examination. For coverage of image acquisition, integrity verification, and containers, refer to Chapter 3. Disks can be analyzed on several logical levels such as the raw bytes, or in terms of partitions, filesystems, or files. At the raw disk level, the most obvious type of analysis is to look at the bytes. Several viewers are available which present the investigator with a hexadecimal encoded view of the raw disk: WinHex [51], Ghex [52], xxd [53], and hexedit [54] are just a few. This type of analysis is helpful when a more powerful tool is not available, but is of limited overall help simply because viewing terabytes of hex-encoded data one page at a time sheds little light on the state of the machine or the events which have occurred on it. Hex viewers do however form a baseline in that they exist for all common platforms, and can be used to view any type of file. Since we are often interested in finding references to people, places, or key words, tools exist which will parse strings resembling words from raw disks. Some simply output the strings found (Binutils *strings* [55]) while others such as dtSearch,[19] which are more full-featured produce highly optimized full-text indexes of strings from multiple encodings (ACSII, Unicode), which facilitate fast searches. As some disks have no filesystems on them, either due to malicious intent, or accidental corruption, investigators have a their disposal *file carvers* (e.g. ,Scalpel [17], Foremost [56], or Rapier [57]). These tools use several techniques in an attempt to recover files from a disk where no filesystem metadata exists, by searching a disk for known binary signatures of specific file types and then copying out sequences of bytes as potential recovered files. Where partitions do exist, tools like *fdisk* [58] can list the partitions on a disk. If filesystems exist within the partitions, tools from the Sleuthkit suite allows an investigator to fully parse files and filesystem metadata, attempt recovery of deleted files, create timelines of filesystem activity, and many other useful operations. At the file level, the *file* tool can determine the type of a file by looking at its structure, and *grep* [59] can search for strings within files. Myriad application-specific tools exist, and are discussed below.

4.1.2 Network Forensics

As more data moves across the network and over the Internet, the need for tools to investigate these data flows increases in importance. tcpdump [60], windump [61], snort [62] and similar can capture into an easily parseable format (such as PCAP [63]) all live traffic that crosses the network interface of a machine. Wireshark [64], a much more powerful tool, can similarly capture network traffic, but can also decode hundreds of common protocols into easily viewable formats, and can reconstruct independent data streams into sessions. Additionally, Wireshark can decrypt SSL /TLS sessions under some circumstances when the needed keys are known. ngrep [65] facilitates string search on network captures, similarly to grep above, but is also network-packet-aware. Tcpextract [66] can extract complete files from network flows in which files were transferred. The dsniff [67] suite of network capture and protocol parsing tools performs a whole host of functions, including ARP cache poisoning, password sniffing for many plaintext protocols, and URL mirroring. Tools like Nmap [68] can scan networks for hosts, and determine the operating system and server applications in use. Netcat [69] is a simple network client and server which allows an investigator to easily move data around a network. Resolving hostnames to IP addresses and the reverse can be done with dig [70], whois [71], and nslookup [72]. While the above tools work on many types of networks, several tools exist specifically for analyzing wireless networks. Kismet [73] and Netstumbler [74] can scan for wireless networks, including those which are hidden, and Kismet can log wireless network traffic.

4.1.3 RAM Forensics

There is increasing interest in performing deep memory analysis as a standard part of digital forensics investigation, because a substantial amount of potential evidence is lost if this source is ignored. Recently, a number of utilities for parsing Windows memory dumps have been developed, with a primary catalyst being the 2005 DFRWS memory analysis challenge [75]. The Volatility framework [76] extracts information from Windows XP SP2 and SP3 memory dumps, including a list of running processes, open network connections, loaded DLLs, and Virtual Address Descriptor (VAD) information. The knttools [77] dump information about processes, threads, access tokens, the handle table, and other OS structures from a Windows memory dump. Memparser [78] is capable of outputting similar information with cross-referencing used to detect hidden objects. Schuster's ptfinder tools [79] take a different approach and instead of walking OS structures, attempt to carve objects that represent threads and processes directly from the memory dump. This allows hidden processes to be more easily discovered and can also reveal information about recently terminated processes. [80] provides an overview of several memory acquisition tools for Microsoft Windows. PyFlag, in addition parsing disks, log files, network traces, parses Windows memory dumps (by incorporating Volatility), and Linux memory dumps for some specific kernels by wrapping the crash [81] kernel debugger. Memoryze [82] can be used to enumerate running processes, loaded drivers and search for hooks on live Windows systems as well as on memory dumps.

4.1.4 Application-Specific Tools

Hundreds of other small single-purpose tools exist, each of which facilitates examination of data specific to some application. Any attempt at an exhaustive discussion is futile, but here we give a sampling of what exists. An investigator can construct timelines of file accesses with mactime, using file MAC times. CacheView [83], Pasco [84], and Web Historian [85] parse browser caches and history for common web browsers. Skype-Parser [86] gives information on when calls and chat sessions occurred, and who the participating parties were. P2P Marshall [87] discovers information on p2p software installed, removed, and used on a system. Paraben's Chat Examiner [88] parses log files from many common chat programs. Process Monitor [89], Access Data Registry Viewer [90], RegViewer [91] and others provide various methods for Windows registry analysis, on both Windows and Linux platforms. NTFS alternate data streams can be viewed with Stream Viewer [92]. Recycle Bin metadata files (index2.dat) can be parsed by rifiuti [93] to show evidence of files previously emptied form the Recycle Bin. Parsers exposing data and metadata for specific file types abound: Metadata Analyzer [94] for Microsoft Office files, libpst [95] for Outlook mail archives, libdbx [96] for Outlook Express dbx files, PEView [97] for Portable Executable Format files, TNEF [98] for ms-tnef MIME attachments, jpeginfo [99] for JPEG files, and pdfinfo [100] for PDF files. Again, this just scratches the surface.

4.1.4 Live Forensics

Live forensics deals with investigating the state of a live system, either while it is currently running or from an image of (some part of) its live state. This changes the difficulty and effectiveness of an investigation in several ways. On a live system, the investigator has access to all of the utilities on the system, and the running operating system itself. On the flipside, however, the state of the system is continuously changing, and old data is constantly being overwritten with new data. Each operation performed to acquire evidence from a live machine potentially destroys other evidence as files are written to disk, processes are loaded into memory, and operating system bookkeeping occurs in the background. That being said, live investigation is sometimes required, and tools and techniques exist to support it. Encase Enterprise edition [101] adds live forensics capabilities for enterprise networks by deploying software agents on machines to be monitored. These agents can capture memory and perform other monitoring activities under the supervision of a forensic analyst. The Mobile Forensics Platform [102], now called the OnlineDFS in its commercial incarnation, allows remote, live investigation of forensics targets without the need to install software agents on the machines under investigation. Administrative credentials are used to retrieve a variety of information about the running system, including process lists, open files, and networking statistics. In a live investigation, standard built-in system utilities become forensically useful for gathering evidence, including: ifconfig, ipconfig, netstat, date, uptime, ps, lsof, dozens of utilities from the sysinternals [103] suite and many others. LiveCD's Incident Responder Collection Report (IRCR) [104], and RAPIER [105] combine the functionality of several system tools in a batch script for easily acquiring necessary volatile information on live system state.

4.1.6 Password Recovery

As computer users become savvier, the use of password protection and encryption has increased. Applications like John the Ripper [106], PRTK, Cain and Abel, and ElcomSoft Password Recovery Bundle [107] offer differing levels of functionality for recovering password for user accounts on multiple operating systems as well as myriad password protected file types (Microsoft Word, WinZip, etc).

4.1.7 Tool Suites

Combining the functions of many of the single purpose tools discussed above, Encase, and FTK are graphical commercial digital forensics suites with the ability to acquire drive images, parse most common file types, create full-text keyword indexes, carve deleted files and perform other common disk analysis tasks. The Sleuthkit, and its graphical interface, Autopsy provide a suite of tools for analyzing disks at all logical levels, as well as filesystem timeline creation. PyFlag is an environment for analyzing data from disks, network captures and system RAM; it contains many log file and protocol parsers.

4.2 Issues

While DFI has available to it a great many tools and techniques as seen above, there are just as many gaping holes in the functionality required to perform truly comprehensive investigations. The reasons for this state of affairs are several. As a field, DFI is relatively new, and there just has not been enough time and manpower devoted to researching appropriate methods and developing effective tools. The set of investigative targets is not static; as new applications and machines are developed, new tools are required. Several fundamental operations cannot be performed yet, some of which are listed here. No tool exists for deep analysis of Linux RAM images across kernel versions and distributions. Carving of fragmented files across all popular file types, with low false positive rates cannot be done. Complete system timelines utilizing all sources of timestamps (from MAC times, registry keys, kernel structures, log files and any other source of timestamps) cannot be created. Classification of audio and video files and the ability to search them for sounds or images does not exist. We do not yet automatically integrate data from multiple sources into a single view of system state. We have the rudimentary ability to parse encryption keys from RAM, but no usable tools for decrypting Skype encrypted logs and captures, SSH sessions in network captures and SSL traffic. These are but a few missing pieces, and these are the simpler ones. More difficult will be developing tools which extract high-level meaning from the data parsed by the tools we have. For example, we have parsers for registry files, but no tools which provide understanding of full forensic implications of the keys in the files. The tools we do have must be constantly updated for new versions of hardware, operating systems, applications, and file formats, or they quickly become impotent. We have little or no ability to do filesystem forensics on newer filesystems such as ZFS [108], or BTRFS [109]. We are just beginning to appreciate the need for techniques for investigating non-traditional computers, such as smartphones, game consoles, and GPS devices. Still beyond us are large single targets, and large networks of targets, due in part to limited resources, but also in part to poor resource utilization; this issue will be addressed in Chapter 5. As more and more user data moves to online social networking sites, we require new techniques for discovery of evidence there. Most tools support Windows and Linux, far fewer support Solaris, *BSD, or OS X, to say nothing of the several popular mobile device operating systems.

In order to address one missing piece of the puzzle and therefore enhance the effectiveness of DFI, we present *ramparser*, a tool for deep parsing of Linux RAM images.

4.3 RAM Analysis

Recent studies have illustrated that data persists for a long time in volatile memory [110][111]. Unlike tools that analyze running machines, such as Encase Enterprise and OnlineDFS, off-line memory analysis tools extract digital evidence directly from physical memory dumps. These memory dumps may be acquired using a number of different mechanisms (dependent on OS type and version), from hardware–based approaches such as Tribble [9] and via Firewire [112] to software–only approaches, such as using dd [113] to access the physical memory device or via insertion of custom kernel modules.

These memory dumping mechanisms are not infallible and some high–tech approaches to subverting memory acquisition have been proposed [114]. Fortunately, unless the subversion mechanism is very deeply embedded in the OS, a substantial amount of overhead may be incurred to prevent acquisition, potentially revealing the presence of a malicious agent [115]. A recently released tool provides another alternative for memory acquisition, by converting Windows hibernation files to usable memory dumps [116]. Finally, a novel approach to memory acquisition called BodySnatcher, involving injection of a small, forensic OS that subverts the running OS, was presented at DFRWS 2007 [117]. Surprisingly, there has been little work in deep parsing of Linux memory dumps. idetect [118] is a proof of concept tool that parses 2.4–series memory dumps and enumerates page frames, discovers user mode processes, and provides detailed information about process descriptors. [119] discusses many of the relevant OS structures that must be parsed to extract digital evidence from Linux memory dumps.

Due to the lack of available memory parsing tools for Linux, we developed *ramparser*, a tool for deep analysis of Linux memory dumps. Specifically, the current version is able to handle a range of 32-bit 2.6 kernel variants. The information provided by *ramparser* from the memory dump includes running processes, open network connections, in kernel socket buffers, loaded kernel modules, and a specific process's memory-mapped and open files, code, and data. For experienced UNIX users, the tool is capable of simulating commands such as *ps*, *lsmod*, and *netstat*. It is also capable of writing out process-specific information and data to files for later investigation.

```
struct list_head {
      struct list_head *next, *prev;
};
```

Figure 4.1: struct *list_head.*

Since the Linux kernel is written in the C programming language much of the interesting data is laid out in linked lists of C structures. In order to reliably walk large amounts of memory and find valid structures, common routines were created to validate specific data structures used often in the kernel. For example, the *list_head* (Figure 4.1) data structure, which implements circularly linked lists in the kernel, contains two members, *next* and *prev*. In order to help debugging, after being freed these members are set to poison values. This creates only two possible values for each of the members, either a valid kernel pointer or their respective poison values. Incorporating knowledge of these special values and others when searching for various data types allows the program find valid structures quickly and with few false positives. The most reliable and useful data types used to validate are lists, enums, stack based character buffers, and kernel pointers which cannot be null. Lists are implemented as *list_head* structures which can be found as discussed above. Enums are useful for eliminating false positives because they generally have a small range of valid integer values, such as -1, 0, or 1. Stack based character buffers are excellent for debugging since they can be easily printed, and validating the strings helps reduce false positives. Kernel pointers that cannot be null also have a relatively small range, i.e., PAGE OFFSET to 0xffffffff on 32–bit systems. Pointer structure members which can take the null value are not useful since RAM generally has many zero filled areas and accounting for them slows down searching. Of limited use are some integers and shorts which should never have negative values.

```
struct task_struct {
      volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
      struct thread_info *thread_info;
      atomic_t usage;
      unsigned long flags;    /* per process flags, defined below */
      unsigned long ptrace;
      int lock_depth;         /* BKL lock depth */
      ...
      struct list_head tasks;
      ...
      struct mm_struct *mm, *active_mm;
      ...
      pid_t pid;
      pid_t tgid;
      ...
      /* PID/PID hash table linkage. */
      struct pid_link pids[PIDTYPE_MAX];
      struct list_head thread_group;
      ...
      /* process credentials */
      uid_t uid,euid,suid,fsuid;
      gid_t gid,egid,sgid,fsgid;
      struct group_info *group_info;
      kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
      unsigned keep_capabilities:1;
      struct user_struct *user;
      ...
      /* filesystem information */
      struct fs_struct *fs;
      /* open file information */
      struct files_struct *files;
      ...
};
```

Figure 4.2: struct *mm_struct.*

In order to determine which processes were running on the system when the RAM capture was created, we must first find the *task_struct* for the process "init." Structure *task_struct* is the kernel representation of a running process. It is a huge structure containing tremendous amounts of information on a single running process, including the process identifier (pid), the run state of the process, and pointers to this processes open files, memory regions, and copious other data (Figure 4.2). Finding *task struct*'s is very reliable since the structure contains an enum for the sleep type, a stack–based character buffer for the process name, many non-null kernel pointers, unsigned integers, and lists. By validating the numerous members of the structure as it walks memory, the program rarely produces false positives. The -d option to the *ramparser* tool will parse the memory image for *task struct*'s and print the relevant members. In Linux operating systems, the father of all processes on the system is the "init" process, generally pid 1. Since we know it must be running, and we know it's name and pid, the first thing that *ramparser* does is scan the memory dump and locate the init process' address by carving *task struct*'s from memory until one is found with a pid of "1". After this, *ramparser* will be able to walk the entire list of active processes. The address of *init_mm* is set next so that we can find paged data in the kernel. The kernel convention is to use *init_mm* as the page directory pointer for any in–kernel data which requires paging instead of being identity mapped. After these values are set, the program parses the user supplied arguments and performs the desired analysis.

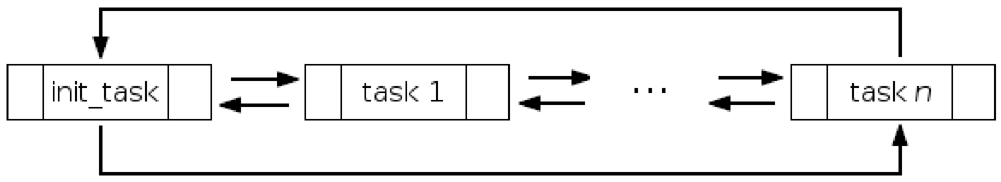
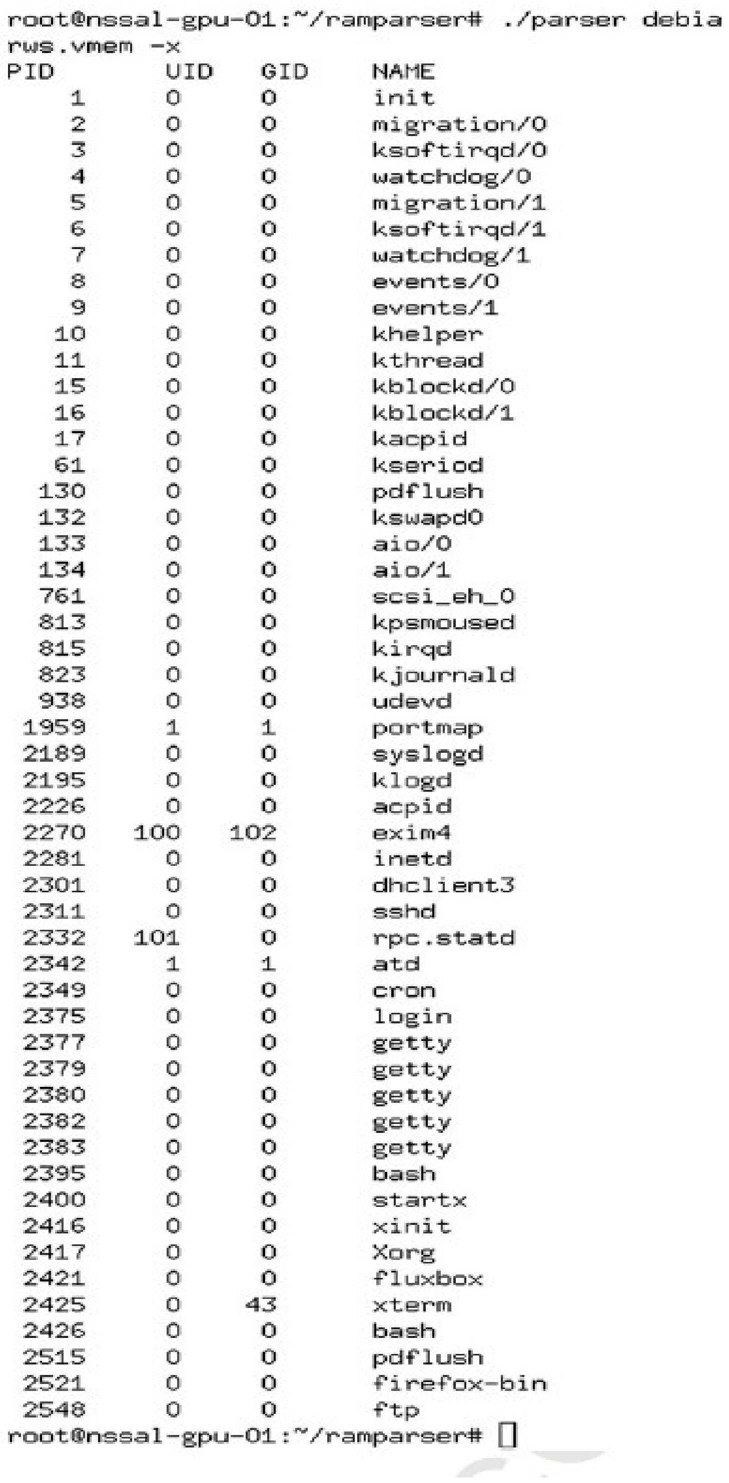


Figure 4.3 Linked List of *task_struct*s

Retrieving Process Information. After finding init it is then possible to walk the *tasks* member of init's *task struct* which holds the linked list of all active tasks (Figure 4.3). Partially simulating the ps(1) command is then a straightforward operation, involving walking the process list and printing out detailed information for each process. More useful operations such as walking a process' memory maps, open files, and network sockets are possible by using similar constructs already used in the kernel. The *ramparser* -x option performs the simple ps(1) operation.
See Figure 4.4 below for a sample process listing.



```
root@nssal-gpu-01:~/ramparser# ./parser debia
rws.vmem -x
PID       UID    GID     NAME
    1       0      0     init
    2       0      0     migration/0
    3       0      0     ksoftirqd/0
    4       0      0     watchdog/0
    5       0      0     migration/1
    6       0      0     ksoftirqd/1
    7       0      0     watchdog/1
    8       0      0     events/0
    9       0      0     events/1
   10       0      0     khelper
   11       0      0     kthread
   15       0      0     kblockd/0
   16       0      0     kblockd/1
   17       0      0     kacpid
   61       0      0     kseriod
  130       0      0     pdflush
  132       0      0     kswapd0
  133       0      0     aio/0
  134       0      0     aio/1
  761       0      0     scsi_eh_0
  813       0      0     kpsmoused
  815       0      0     kirqd
  823       0      0     kjournald
  938       0      0     udevd
 1959       1      1     portmap
 2189       0      0     syslogd
 2195       0      0     klogd
 2226       0      0     acpid
 2270     100    102     exim4
 2281       0      0     inetd
 2301       0      0     dhclient3
 2311       0      0     sshd
 2332     101      0     rpc.statd
 2342       1      1     atd
 2349       0      0     cron
 2375       0      0     login
 2377       0      0     getty
 2379       0      0     getty
 2380       0      0     getty
 2382       0      0     getty
 2383       0      0     getty
 2395       0      0     bash
 2400       0      0     startx
 2416       0      0     xinit
 2417       0      0     Xorg
 2421       0      0     fluxbox
 2425       0     43     xterm
 2426       0      0     bash
 2515       0      0     pdflush
 2521       0      0     firefox-bin
 2548       0      0     ftp
root@nssal-gpu-01:~/ramparser# []
```

Figure 4.4: Process Listing.

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
      struct mm_struct * vm_mm;     /* The address space we belong to. */
      unsigned long vm_start;       /* Our start address within vm_mm. */
      unsigned long vm_end;         /* The first byte after our end address
                                 within vm_mm. */
      /* linked list of VM areas per task, sorted by address */
      struct vm_area_struct *vm_next;
      ...
      /* Function pointers to deal with this struct. */
      struct vm_operations_struct * vm_ops;
      ...
};
```

Figure 4.5 struct *vm_area_struct*

Finding mapped files. Under Linux, virtually contiguous mapped regions with the same permissions are represented by a *vm_area_struct* (Figure 4.5). These represent a process' stack, heap, code section, data section, the data and code section of shared libraries, shared memory, and anonymously mapped memory. These structures can be viewed on a running machine by executing 'cat /proc/<pid>/maps' for the process of interest. *ramparser*'s -p options simulates the maps file for a process by walking the list of *vm_area_struct* structures that are contained within the process' *mm_struct* and printing the starting and ending address, permissions, and the mapped file's name, if available. The -v option of *ramparser* will write the memory pages covered by a processes *vm_area_struct*'s to disk. Since these areas are paged, each memory region is handled 4096 bytes a time when determining the offsets of the data. Using these offsets, an investigator can completely recreate the running process at the time the memory image was taken. See Figure 4.6 for sample output for an FTP process.

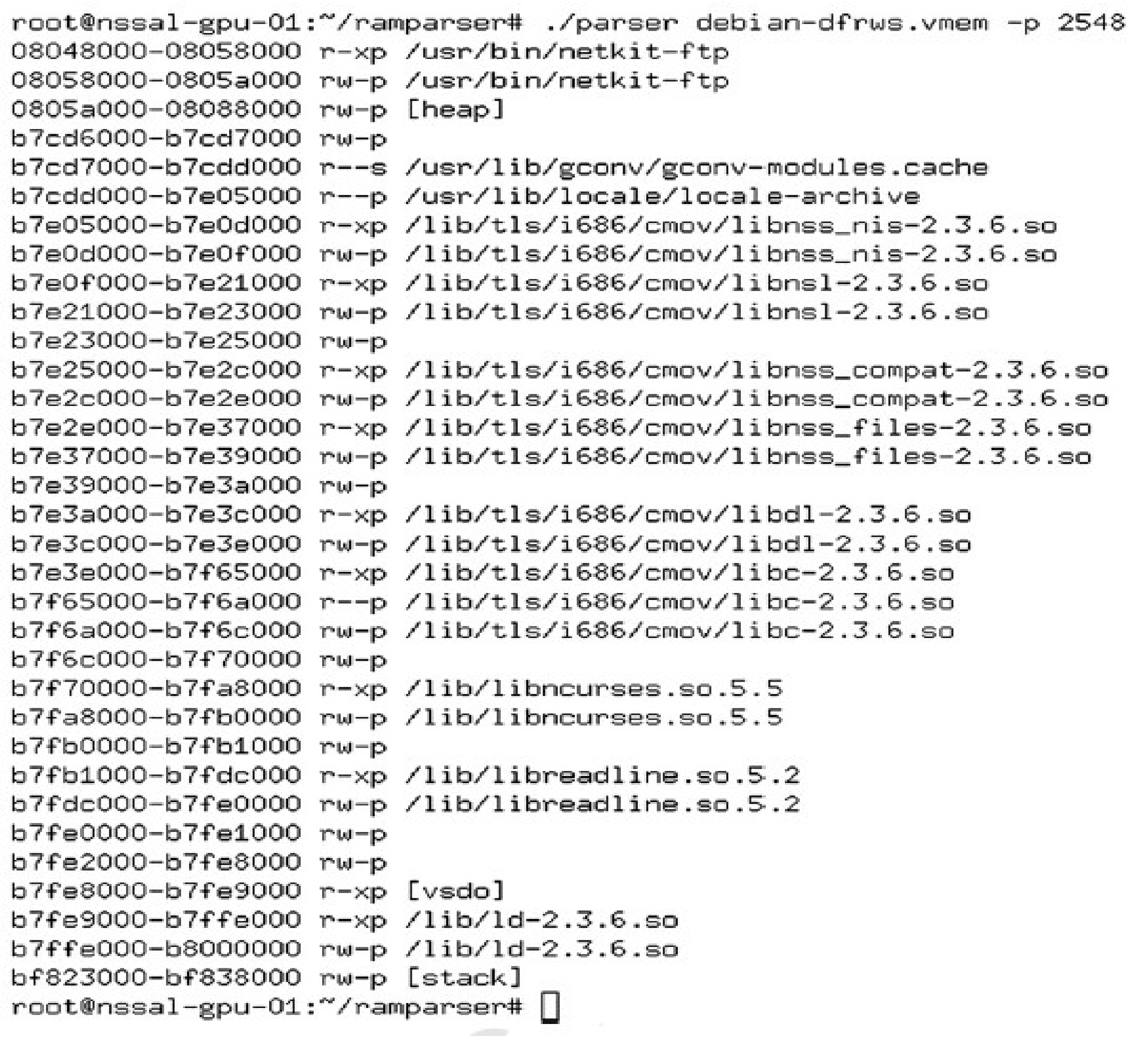

```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -p 2548
08048000-08058000 r-xp /usr/bin/netkit-ftp
08058000-0805a000 rw-p /usr/bin/netkit-ftp
0805a000-08088000 rw-p [heap]
b7cd6000-b7cd7000 rw-p
b7cd7000-b7cdd000 r--s /usr/lib/gconv/gconv-modules.cache
b7cdd000-b7e05000 r--p /usr/lib/locale/locale-archive
b7e05000-b7e0d000 r-xp /lib/tls/i686/cmov/libnss_nis-2.3.6.so
b7e0d000-b7e0f000 rw-p /lib/tls/i686/cmov/libnss_nis-2.3.6.so
b7e0f000-b7e21000 r-xp /lib/tls/i686/cmov/libnsl-2.3.6.so
b7e21000-b7e23000 rw-p /lib/tls/i686/cmov/libnsl-2.3.6.so
b7e23000-b7e25000 rw-p
b7e25000-b7e2c000 r-xp /lib/tls/i686/cmov/libnss_compat-2.3.6.so
b7e2c000-b7e2e000 rw-p /lib/tls/i686/cmov/libnss_compat-2.3.6.so
b7e2e000-b7e37000 r-xp /lib/tls/i686/cmov/libnss_files-2.3.6.so
b7e37000-b7e39000 rw-p /lib/tls/i686/cmov/libnss_files-2.3.6.so
b7e39000-b7e3a000 rw-p
b7e3a000-b7e3c000 r-xp /lib/tls/i686/cmov/libdl-2.3.6.so
b7e3c000-b7e3e000 rw-p /lib/tls/i686/cmov/libdl-2.3.6.so
b7e3e000-b7f65000 r-xp /lib/tls/i686/cmov/libc-2.3.6.so
b7f65000-b7f6a000 r--p /lib/tls/i686/cmov/libc-2.3.6.so
b7f6a000-b7f6c000 rw-p /lib/tls/i686/cmov/libc-2.3.6.so
b7f6c000-b7f70000 rw-p
b7f70000-b7fa8000 r-xp /lib/libncurses.so.5.5
b7fa8000-b7fb0000 rw-p /lib/libncurses.so.5.5
b7fb0000-b7fb1000 rw-p
b7fb1000-b7fdc000 r-xp /lib/libreadline.so.5.2
b7fdc000-b7fe0000 rw-p /lib/libreadline.so.5.2
b7fe0000-b7fe1000 rw-p
b7fe2000-b7fe8000 rw-p
b7fe8000-b7fe9000 r-xp [vsdo]
b7fe9000-b7ffe000 r-xp /lib/ld-2.3.6.so
b7ffe000-b8000000 rw-p /lib/ld-2.3.6.so
bf823000-bf838000 rw-p [stack]
root@nssal-gpu-01:~/ramparser#
```

Figure 4.6: Sample Process Mapping.

```
struct files_struct {
  /*
   * read mostly part
   */
      atomic_t count;
      struct fdtable *fdt;
      struct fdtable fdtab;
  /*
```

```
    * written part on a separate cache line in SMP
    */
       spinlock_t file_lock ____cacheline_aligned_in_smp;
       int next_fd;
       struct embedded_fd_set close_on_exec_init;
       struct embedded_fd_set open_fds_init;
       struct file * fd_array[NR_OPEN_DEFAULT];
};
```

Figure 4.7: struct *files_struct.*

Finding open files. The process descriptor contains a *files_struct* (Figure 4.7) structure which includes a *struct_fdtable* which holds an array of *struct_file* structures. By following these links, it becomes easy to traverse all the file descriptors of a process. Each file descriptor is represented by a *struct_file* which *ramparser* uses to extract the open files, their permissions, and file descriptor number. Files with forensics interest include open files on disk, pipes, and sockets. This information can be viewed on a running system by executing 'ls -l /proc/<pid>/fd'. *ramparser* simulates this functionally in the -o option by walking the file descriptor array and printing the file descriptor number and name of the file opened. Filenames for files on disk are simply their full pathnames, while names for sockets and pipes are formed by joining socket[<inode number>] and pipe[<inode number>] respectively. See Figure 4.8 for sample output for an FTP process.

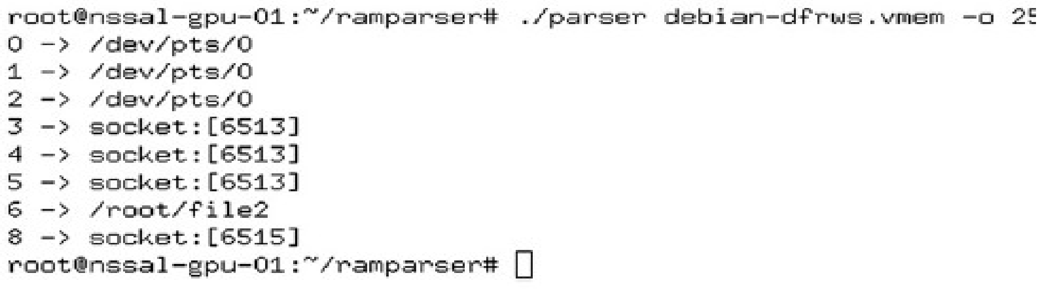
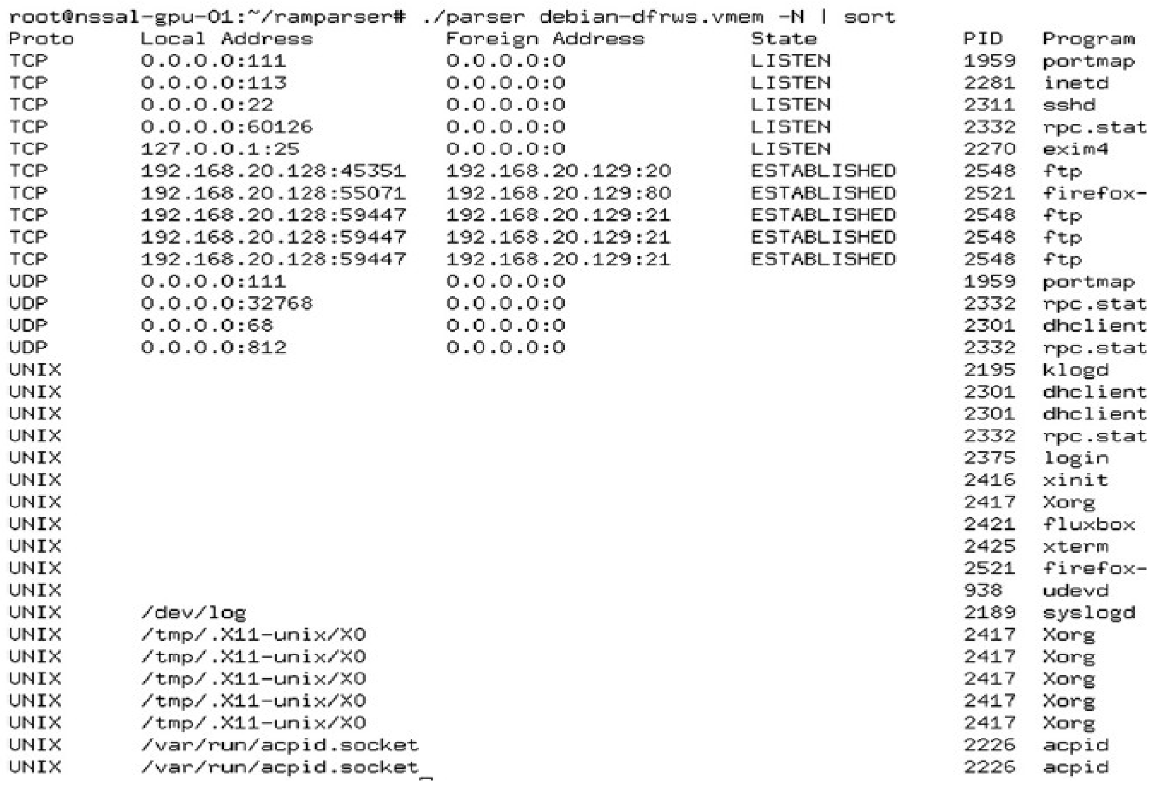

```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -o 25
0 -> /dev/pts/0
1 -> /dev/pts/0
2 -> /dev/pts/0
3 -> socket:[6513]
4 -> socket:[6513]
5 -> socket:[6513]
6 -> /root/file2
8 -> socket:[6515]
root@nssal-gpu-01:~/ramparser# 
```

Figure 4.8: Sample Open Files Listing.

```
struct socket {
      socket_state            state;
      unsigned long           flags;
      const struct proto_ops  *ops;
      struct fasync_struct    *fasync_list;
      struct file       *file;
      struct sock       *sk;
      wait_queue_head_t wait;
      short             type;
};
```

Figure 4.9: struct *socket.*

Finding sockets/netstat information. Since UNIX systems treat sockets as file descriptors, information about open network connections can be gathered by analyzing a process' socket file descriptors. Each socket is represented by a *struct_socket* (Figure 4.9) which contains a pointer to the *struct_sock* for the socket. The socket structure contains the socket's family, protocol, receive and send queues, and state. The *inet_sock* structure representation of *struct_sock* gives the source and destination addresses and ports. Using this information it is possible to implement a netstat(8) like functionality. *ramparser* simulates netstat for all sockets when run with the -N option or gives information about a single process when run with -n. See Figure 4.10 for sample netstat output.



```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -N | sort
Proto     Local Address          Foreign Address        State            PID   Program
TCP       0.0.0.0:111            0.0.0.0:0              LISTEN           1959  portmap
TCP       0.0.0.0:113            0.0.0.0:0              LISTEN           2281  inetd
TCP       0.0.0.0:22             0.0.0.0:0              LISTEN           2311  sshd
TCP       0.0.0.0:60126          0.0.0.0:0              LISTEN           2332  rpc.stat
TCP       127.0.0.1:25           0.0.0.0:0              LISTEN           2270  exim4
TCP       192.168.20.128:45351   192.168.20.129:20      ESTABLISHED      2548  ftp
TCP       192.168.20.128:55071   192.168.20.129:80      ESTABLISHED      2521  firefox-
TCP       192.168.20.128:59447   192.168.20.129:21      ESTABLISHED      2548  ftp
TCP       192.168.20.128:59447   192.168.20.129:21      ESTABLISHED      2548  ftp
TCP       192.168.20.128:59447   192.168.20.129:21      ESTABLISHED      2548  ftp
UDP       0.0.0.0:111            0.0.0.0:0                               1959  portmap
UDP       0.0.0.0:32768          0.0.0.0:0                               2332  rpc.stat
UDP       0.0.0.0:68             0.0.0.0:0                               2301  dhclient
UDP       0.0.0.0:812            0.0.0.0:0                               2332  rpc.stat
UNIX                                                                     2195  klogd
UNIX                                                                     2301  dhclient
UNIX                                                                     2301  dhclient
UNIX                                                                     2332  rpc.stat
UNIX                                                                     2375  login
UNIX                                                                     2416  xinit
UNIX                                                                     2417  Xorg
UNIX                                                                     2421  fluxbox
UNIX                                                                     2425  xterm
UNIX                                                                     2521  firefox-
UNIX                                                                     938   udevd
UNIX      /dev/log                                                       2189  syslogd
UNIX      /tmp/.X11-unix/X0                                              2417  Xorg
UNIX      /tmp/.X11-unix/X0                                              2417  Xorg
UNIX      /tmp/.X11-unix/X0                                              2417  Xorg
UNIX      /tmp/.X11-unix/X0                                              2417  Xorg
UNIX      /tmp/.X11-unix/X0                                              2417  Xorg
UNIX      /var/run/acpid.socket                                          2226  acpid
UNIX      /var/run/acpid.socket_                                         2226  acpid
```

Figure 4.10: Sample netstat Output.

```
struct sk_buff {
      /* These two members must be first. */
      struct sk_buff          *next;
      struct sk_buff          *prev;

      struct sock       *sk;
      struct skb_timeval      tstamp;
      struct net_device *dev;
      struct net_device *input_dev;

      union {
            struct tcphdr     *th;
            struct udphdr     *uh;
            struct icmphdr    *icmph;
            struct igmphdr    *igmph;
            struct iphdr      *ipiph;
            struct ipv6hdr    *ipv6h;
            unsigned char     *raw;
      } h;
      ...
```

```
	/* These elements must be at the end, see alloc_skb() for details.  */
	unsigned int		truesize;
	atomic_t		users;
	unsigned char		*head,
				*data,
				*tail,
				*end;
};
```

Figure 4.11: struct *sk_buf.*

Finding network buffers. Most network investigations involve packet captures taken from the hostile network after suspicious activity is detected. By using the internal kernel network structures it is possible to get even more information related to the network activity at the time of the memory dump. A network socket buffer is represented by *struct_sk_buff* (Figure 4.11) which contains protocol–specific information and points to the beginning and end of a complete packet. Inside each *struct_sock* structure is a receive queue and a send queue of socket buffers which hold data yet to be processed. By collecting these queues from open sockets, data that is yet to be sent out or data that is yet to be processed by a userland application can be gathered and associated with a specific process. Our experiments revealed that the receive queue was usually empty since userland servers process data very quickly, which removes the buffers from the queues. Unlike the receive queues, however, the send queues were generally full during large file transfers. Tests were run uploading files through FTP to outside networks, and *ramparser* was able to recover large parts of the files being transferred. *ramparser*'s -k and -q options can be used to dump the send and receive queues of a process to files.

```
struct module
{
	enum module_state state;
	/* Member of list of modules */
	struct list_head list;
	/* Unique handle for this module */
	char name[MODULE_NAME_LEN];
	...
	/* Startup function. */
	int (*init)(void);
	...
	/* Here are the sizes of the init and core sections */
```

```
        unsigned long init_size, core_size;
        /* The size of the executable code in each section.  */
        unsigned long init_text_size, core_text_size;
        ...
};
```

Figure 4.12: struct *module.*

Finding loaded modules. Loadable modules allow users to insert code dynamically into a running kernel. While this has obvious advantages, it is also a very common entry point for rootkits and other malware to run kernel-level code. Modules are represented in the kernel by a *struct_module* (Figure 4.12) which is defined in include/linux/module.h. *ramparser* is able to find loadable modules in memory with rare false positives due to the module structure containing an enum, list, stack based character buffer, and many kernel pointers. Searching for modules is the slowest part of the code since many of the pointers can be null. This decreases the performance of address validation. After locating a valid struct it is possible to recreate output obtained from the lsmod(8) command as viewed on a running Linux machine.

4.4 Discussion

Performing thorough DFI constantly requires new tools and techniques to fill in the gaps between what we would like to be able to analyze and what we can currently analyze. In this chapter we discussed comprehensiveness in DFI and presented *ramparser* a tool for deep analysis of Linux memory. *Ramparser* fills a gap in the set of functionality currently offered by digital forensics tools. Using an image of physical ram, it parses kernel data structures in order to shed light on the live state of the system as of when the image was taken. The data provided is from the volatile state of the running system and would otherwise be lost. *ramparser* gives the investigator access to the list of running processes on the system, including parsing each processes stack, heap, and memory maps, and enumerating which files and network connections the process has open. This facilitates deep investigation of the functions of these processes. It also lists modules which were loaded into the kernel; these are worthy of investigation as they are a common entry point for malware. Last, *ramparser* can list all files which were open on the system, and all network connections. This information gives further insight into the state of the running system. These functions duplicate the functionality of several useful utilities that run on a live machine, including ps for process listings, netstat for network connections, lsof for open files, and lsmod for loaded modules. This tool aids in conducting more effective DFI by facilitating a more comprehensive investigation. In the next chapter we turn to the issue of efficiency in DFI.

Chapter 5:
Efficiency

The ability to conduct digital forensic investigations (DFIs) within the constraints of available time, processing power and storage capacity is of paramount importance. This is a difficult problem, as optimizing the use of one of these resources generally has the opposite effect on the other resources in the system. It is easy to see that we could perform investigations with minimal computational resources if we can use an eternity of time. Similarly, we could perform investigations extremely rapidly with infinite computational resources. Clearly neither of these is an acceptable tradeoff. We do however require tools to be as efficient as possible for at least three reasons. With more efficient tools we can increase investigative throughput, which would alleviate current case backlogs. We can bring more tools to bear on a single investigation, if each was more efficient. Last, there are targets which are too large to effectively investigate, in terms of large amounts of storage on a single host, or of large numbers of hosts targeted in a single investigation. More efficient tools can facilitate investigation of these currently-out-of-reach targets.

It is clear that terabyte size cases are here and that their investigation is significantly hindered by the inefficiency of the current generation of tools. What may not be readily apparent is that much larger targets are already here. As part of a recent investigation into the leak of plans for operations in Iraq, the DOD seized over 60TB of data [135]. Similarly, as part of the Enron fiasco, 31TB of data were seized. Only the largest law enforcement agencies have access to even enough storage space to acquire that amount of data, much less do any type of investigation. Cases of this magnitude will become more common over time and it is imperative that tools exist which can aid investigators in their task.

5.1 Current Trends

In order to better understand why techniques for more efficient DFI are needed, the next sections discuss three trends that greatly affect the length of time required to conduct effective DFI: growth in the number and size of cases seen, changes in the types of processing power available, and the relationship between the size and IO bandwidth of hard drives.

	Examinations	**Data Processed**	**Average Case Size**
FY 2003	987	82TB	83GB
FY 2008	4524	1756TB	388GB

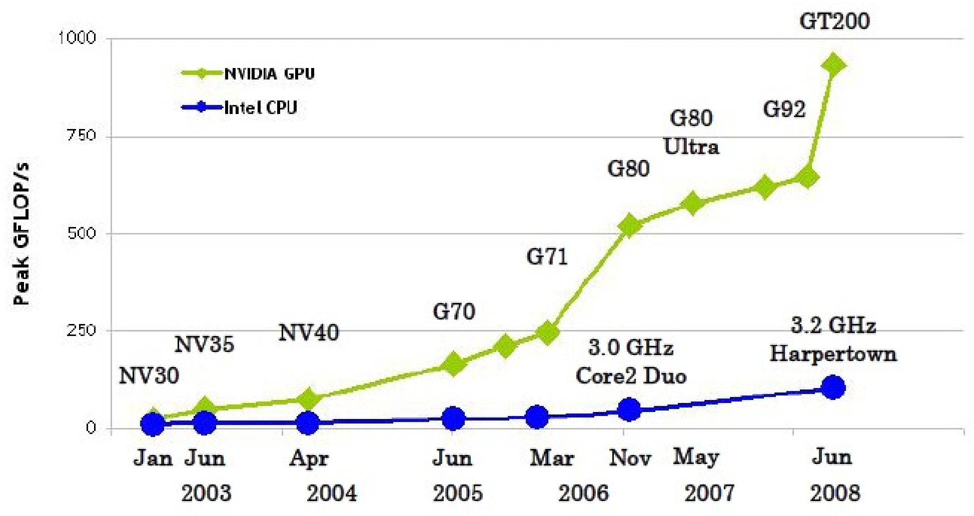
Table 5.1: Case Trends.

5.1.1 Case Trends

First we look at trends in the number, size, and complexity of cases under investigation. In Table 5.1, taken from the FBI [120], we see that the number of cases has more than quadrupled between 2003 and 2008. Even more telling, the amount of data processed during these investigations has seen a greater than 20 times increase in the same period. Average case size has more than quadrupled as well from 83 GB to 388 GB. This rate of growth makes clear the need to use the investigators time as efficiently as possible. As if this wasn't problem enough, individual case complexity is on the rise. In the not too distant past, there was no need to spend time investigating voice over IP traffic, chat messenger logs, local social networking website caches, or mobile phone browser caches, because they did not exist. As new technologies are created, so is the need to be able to analyze the data created. This further increases the computational resources required to perform comprehensive DFIs

5.1.2 Trends in Computational Power

Concurrent with the increases in the number and size and complexity of digital forensic cases, is the changing nature of how computational power is being increased. Up until recently, creating faster machines generally entailed creating processors with faster clock speeds. That paradigm has changed. Clock speed increases have slowed, and instead, multiple processing cores are the vehicle for sustained increases in computational power, as with Intel's new i7 4 core, 8 thread processor [121]. Unlike for faster single cores, taking advantage of the power of multiple cores requires programs be specifically coded to do so. Due to the additional programming complexity of multi-threaded programs and the desire to get a working prototype as quickly as possible, most forensics tools are not multi-threaded, and therefore do not utilize multiple cores. Some tools have recently begun to take advantage of multiple cores on an ad-hoc basis, but with 100-core processors on the near horizon [122], current practices need to be re-thought.



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Figure 5.1: CPU vs. GPU GFLOPS [123].

Another trend on the processor front is the general purpose utilization of highly massively parallel co-processors, such as graphics processing units, termed GPGPU. The 200-series of GPUs from NVIDIA have as many as 240 cores and are capable of over 900 GFLOPS as compared to around 120 GFLOPS for the quad-core 3.2 GHz Intel Harpertown processor [123]. The Compute Unified Device Architecture (CUDA [124]), from NVIDIA is the programming SDK / runtime for writing general purpose programs which can execute on many of their newer GPUs. This computational power made available is not without cost, however. The graphics processor is only useful for highly parallel computations due to its single-instruction multiple-thread (SIMT) programming model. Also difficult to utilize is its complex, multi-tiered memory hierarchy. This makes porting code to the devices even more difficult (and less utilized) than multi-core processors.

A brief description of CUDA is provided here. Complete details are available in [123]. CUDA supports a restricted extension of the C programming language. The few syntax extensions exist to clearly demarcate which instructions are to be executed on the device (GPU) as opposed the host CPU(s), and to specify where in the multi-tiered memory hierarchy of the GPU to allocate storage. Functions to be executed on the device are organized into *kernels*. *Kernels* initialize a user-specified number of threads, organized into *blocks*. *Blocks*, in turn, are organized into a *grid*. While executing on the device, several restrictions exist. The GPU runtime does not support recursion or function pointers, and most of the C standard library functions are unavailable.

Architecturally, the GTX 280 (a representative model from the 200-series), has 30 streaming multi-processors with 8 cores each, giving 240 cores total, and 1 GB of device RAM. Several other memory spaces of varying size and speed exist including globally accessible constant memory and texture memory, and per-multiprocessor shared memory and registers. The difficulty in CUDA programming is a factor of working within the SIMT model, managing blocks and threads, making careful use of the memory hierarchy, and navigating the device's many performance-affecting nuances. More detail is provided in [7].

5.1.3 Storage Trends

Hard disk drives have always been a bottleneck in forensic processing due to the ratio of capacity to IO bandwidth of the devices. The newest high capacity 2TB drives from Western Digital [125] are capable of sustained reads of about 76 MB/s [126]. Given these numbers, it would require about 7.5 hours just to read the entire drive from beginning to end. These numbers are for sequential access, and can increase exponentially if the reads occurs in a random access fashion. While solid state drives are significantly faster, their price per gigabyte will ensure that forensics investigators will have to work with spinning platter hard drives for some time to come.

5.2 Issues

These three trends pose significant problems for DFI for a few reasons. First, most current digital forensics tools do not take advantage of new hardware advancements, such as multi-core processors and massively parallel co-processors, like graphics processing units (GPUs). Further, they do not optimize IO operations, which is critical when one takes into account the increasing size of the investigations being pursued. Current hardware trends are only making things worse. Hard drive capacity is increasing at a much greater rate than both the IO bandwidth of these devices, and the speed of the tools which process the data. Seen from another angle, newer solid state drives provide opportunity for more efficient DFI. Intel's X-25M [127] can sustain reads at over 220 MB/s. Unfortunately, the current generation of tools cannot process data so quickly Similarly, the switch from increasing processor clock speeds to increasing the number of processing cores as the main method for increasing the computational power of these devices [128] requires new coding practices.

5.3 Efficiency Techniques

The efficiency techniques we present and implement are of two distinct types: those related to IO and those related to computation.

5.3.1 Sequential IO

Modern hard disk drives are composed of spinning platters with magnetically encoded data. Data is read from and written to the platters by a set of movable heads. Data is laid out contiguously on the physical platters. This setup makes reading and writing of data sequentially significantly faster than reading from non-contiguous parts of the drive, since expensive seek operations, which require movement of the drive heads, are avoided. Files stored on disk in blocks and filesystems implement algorithms which attempt to lay out these blocks of individual files sequentially on disk. Though these algorithms work well, file fragmentation still increases over time. In order to more efficiently process these operations, all IO should be performed sequentially. By using knowledge of where on disk the reads and writes will be performed, we can order these operations so that the set of all operations takes place in the order in which the blocks appear on disk. Operations on groups of files tend to lump the files according to logical groupings, e.g. compute the MD5 hash of all files in some directory. This logical grouping of files in a directory has no bearing on where these files are laid out on the physical disk. This disparity leads to a huge inefficiency for these types of operations on groups of files. Approaching the operations on the group of files as a single set of operations and ordering them according to their order on disk effects significantly more efficient IO.

5.3.2 Reduce Total IO

As the input data set can be huge, so can the output data set. In some cases, where result sets can include false positives, the output data set can be larger than the input. Due to constraints on storage space and IO bandwidth, efficient tools must minimize the number and size of reads and writes that must occur. This can be accomplished in several ways depending on the specific application. Using compression techniques can reduce the size of the output written size (at some computational expense). Similarly, reduction of the amount of data written (through careful screening during evidence analysis to avoid writing irrelevant data) can improve performance. Using efficient algorithms like Boyer-Moore string search can be sub-linear in their operation and so not require all of the input data to be examined. In the simplest case, many programs are careless in their IO usage, making multiple redundant copies of the same data. Eliminating this waste also reduces the total IO which must be performed. In many cases it may not be necessary to write out all results when data about the location of the result in the original data set allows for on-demand reading of the data only when required by the investigator.

5.3.3 Non-Blocking IO

Digital forensics tools often deal with large datasets which cannot be stored in their entirety in RAM. Traditional single threaded tools generally do the bulk of their work in a loop which reads in a chunk of the input data from disk into RAM, performs some processing on the chunk, and outputs results for that chunk from input to disk. Because of the single threaded nature of the system, each of these three steps occurs in succession. Specifically, while input is being read, no processing is taking place, and no output is being written. Similarly, while processing is taking place, no input is being read, and no output written. This is highly inefficient due to the fact that even on single processor systems data can be read from or written to disk while processing is occurring. This is facilitated by Direct Memory Access (DMA), which allows reading and writing between RAM and certain other subsystems, such as hard drives, without the use of the CPU. In order to take advantage of this facility, DFI tools must perform Non-Blocking IO. We define Non-Blocking IO as that which allows processing to continue before the completion of the IO transaction. In such a system, the three steps in the loop above can occur simultaneously. While the data for step n is being processed, the data for step n+1 can be fetched, and the results from step n-1 can be stored. Here the CPU and IO throughput can be more efficiently utilized.

5.3.4 Multicore Processors

Efficient tools must make use of multiple processor cores. Storage capacity and associated case size are growing at a far greater rate than the processing speed of individual processor cores. Libraries such as PThreads for C, or built-in threading primitives in Java, Python, and other languages must be utilized to speed the processing of parallelizable work. Care must be taken to choose an appropriate language and threading model, as sometimes resulting performance statistics are unintuitive. For example, multithreaded CPU-bound programs written in Python can actually perform more slowly on multicore hardware than single core hardware [129] due to language implementation peculiarities.

5.3.5 Massively Parallel Co-Processors

Co-processors such as GPUs represent a tremendous amount of computational power, and now have significantly more usable APIs and runtimes to facilitate general purpose processing than in the past. Combined with their now widely installed base, their use by DFI tools is necessary to their efficient execution – a resource which begs for utilization.

Until recently, general purpose programming for GPUs was both difficult and typically targeted at very specific problems. To perform non-graphical calculations required techniques that recast data as textures or geometric primitives and expressed the calculations in terms of available graphics operations. Other difficulties included non-IEEE compliant floating point representations, the lack of integer arithmetic, and lack of support for random memory writes. Newer GPUs, including the Gxx series from NVIDIA and the R600 from ATI, solve most of these problems and in addition, have large numbers of scalar processors and excel at executing massively threaded algorithms. In addition, both NVIDIA and ATI have released high-level SDKs for general purpose GPU programming. NVIDIA's SDK is CUDA, which allows NVIDIA GPUs to be programmed using a restricted subset of C. ATI's GPU programming SDK is Brook+ [130], based on Brook, which is itself an extension to C that was originally developed at Stanford University for parallel computation [131]. While most of the existing work on accelerating security applications is currently based on NVIDIA's CUDA, the OpenCL [132] platform, offering execution across NVIDIA and ATI GPUs along with multicore CPUs and other types of processors appears poised to be the dominant player in the future.

Even using the new SDKs for general purpose GPU programming, such as CUDA and Brook+, programmers face a relatively steep learning curve and must also pay close attention to the architectural details of the GPU. Achieving maximum performance when offloading computations onto modern GPUs still requires careful attention to a number of details. One important detail is that modern GPUs use a Single Instruction Multiple Thread (SIMT) execution model, a slightly less restrictive version of the more widely known Single Instruction Multiple Data (SIMD) model. Another is that transfer bandwidth between the host and the GPU is limited by bus speed, mandating that GPU computations have significant "arithmetic intensity" to offset the cost of moving data between the host and GPU. Finally, modern GPUs have complicated memory hierarchies, with available memory divided into a number of classes with varying sizes, caching strategies, and access costs. Despite these difficulties, the effort associated with GPU programming is worthwhile and proper utilization of GPUs can have a positive impact on the performance of DFI tools.

In order to demonstrate the use of these techniques on real-world problems, we apply them to the file carving problem, described in the next section.

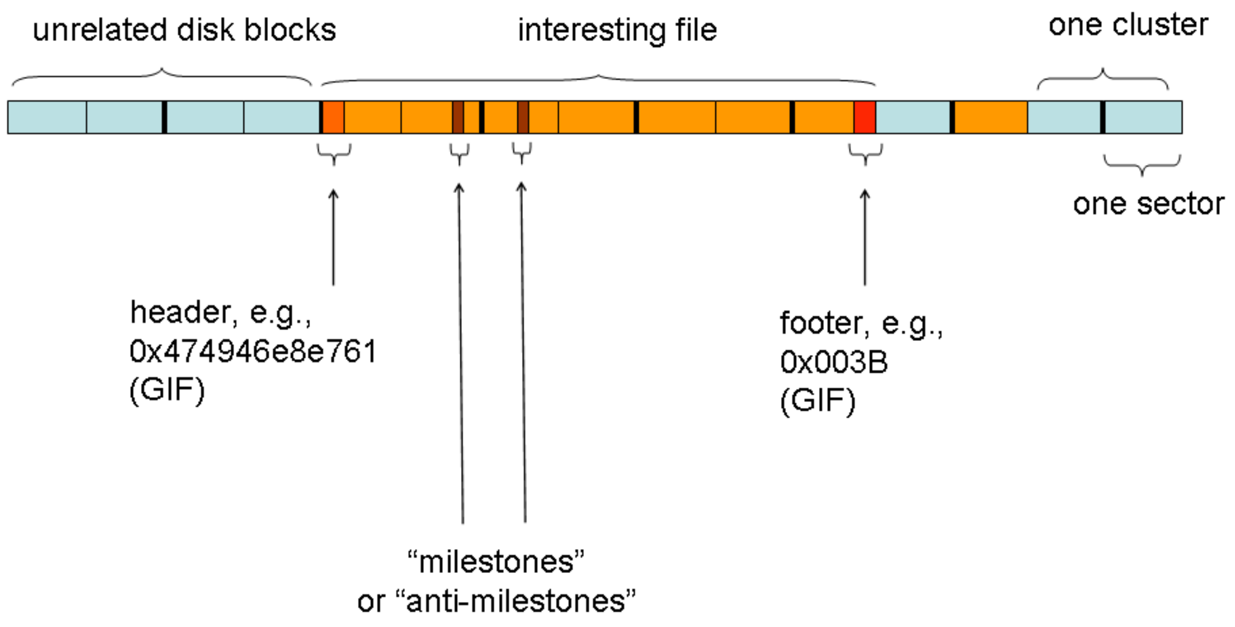

unrelated disk blocks
interesting file
one cluster
one sector
header, e.g., 0x474946e8e761 (GIF)
footer, e.g., 0x003B (GIF)
"milestones" or "anti-milestones"

Figure 5.2: File Carving.

5.4 File Carving

File carving is a technique for recovering files which are no longer properly allocated by an existing filesystem. This can be due to deletion from the filesystem, destruction of the filesystem, or other corruption. File carving, at its most basic level, uses a database of known sequences of bytes which signal the beginning of a file ("header") or end of a file ("footer"). For example, ZIP archives begin with the byte sequence "PK\x03\x04" and end with the byte sequence "\x3C\xAC" (see Figure 5.2). Using this information we can scan a raw disk for occurrences of these byte strings and copy, or "carve," out bytes between headers and footers as potential recovered files. Some types of files require that the footer be included in the file, some require the opposite. Also, for some file types, the footer used must be the farthest away from the header (within some limit) rather than the nearest to the header. This case comes about when the file footer appears more than once in the file. File carving is powerful in that it works on any type of media, and does not depend of the type, or even existence of a filesystem. It does have a significant weakness, a high false positive rate. This occurs for several reasons. First, this method assumes that any instance of a header byte sequence is the beginning of a file of that type, which is clearly not the case. This is especially true for file types with short headers. Also, because files on disk can be stored in non-contiguous chunks of disk, or "fragmented," simply copying the bytes between header and footer will not guarantee that the entire file is captured, or that extra bytes are not included in the files, or that a file even exists there at all – the header and footer found may not belong to any file, much less the same file. There has been a great deal of research on methods for reducing the false positive rate using extensions to this basic version of file carving.

The next simplest extension is to determine file length based on more information than just the relative position of headers and footers in the data stream. Some file types store the length of the file in a field of the internal file metadata. As the point of carving files is generally to open the files with an appropriate viewer to examine its contents, and some viewers for some file types are sensitive to extraneous bytes at the end of the file, using the correct length as specified inside the file itself can help generate more correct carved files.

It is important to note that some file types have complex, well defined internal structures (ZIP, Microsoft Office, PDF and others) and some have no internal structure (plain text files). A greater level of internal structure has more information in it to use for low false positive rate carving, but also must be recovered correctly in order for its contents to be examined. For example, a carved plain text file which is actually incomplete, or contains pieces of other text files, will still be readable in any plain text viewer. A similarly carved ZIP file will not be readable by a ZIP utility, as the internal structure is broken.

In order to further reduce the number of false positives, some tools attempt to do deeper parsing of specific file types. Following with our ZIP file example, the internal structure of files compressed according to the ZIP specification [133] have some metadata about the zip file itself just following the header. This is followed by the compressed versions of each file in the archive. At the end of the file is a directory which details data about each file in the archive, as well as the position of the file in the archive. Using all of this data, one can significantly more accurately carve zZIP archives,and other well-specified file types with internal structure, from raw byte streams. A similar techniques is to use simple header-footer based carving, but then use external file type verifiers to identify the false positives from correctly recovered files.

None of the above techniques help with one of the main difficulties in file carving, fragmented files. Accordingly, much of the file carving research of late has been concerned with developing methods for combating the problem of recovering fragmented files. In [137] it is noted that most fragmented files consist of exactly 2 fragments, often separated by short gaps, and an algorithm is presented to recover these bi-fragmented files. In [138], a method is presented for discovering the non-fragmented beginning of a file, the "base fragment," and then reassembling the entire file using sequential hypothesis testing. Other methods use statistical characteristics inherent in specific types of files, such as documents [143] and JPEG images files [139][140] [141] [142] to correctly carve fragmented files. A broad solution the the general problem of fragment reassembly is still an open research direction.

File carving is representative of a large class of DFI techniques, with which it shares several characteristics. It can take a long time to run even on modest data sets. The input data set can be huge, on the order of terabytes. Storage requirements can be huge. The output set can be as large or significantly larger than the input data set, because of false positives. Depending on the number of file types searched for and the method used to detect them, file carving can be either IO-bound or CPU-bound. It is a technique used in most any investigation. It is rife with possibilities for parallelism. Current file carvers are no designed with efficiency as a concern, in terms of the techniques discussed above. It is with these characteristics in mind that we present a detailed study of the application of said efficiency techniques to the open-source file carver, Scalpel.

5.5 Case Study: Scalpel

In order to demonstrate the effectiveness of the above techniques for efficiency in DFI tools, we will apply a subset of them to the Scalpel file carver. Scalpel is a performance optimized fork of Foremost 0.69, which performs carving based strictly on headers and footers. As our baseline we use a modified version of the widely used and open source Scalpel-1.60. The modification fixed an efficiency bug in the header footer queue algorithm.

Scalpel execution begins by reading in a configuration file containing header, footer and carving rules for each type of file it can recover. Headers and footers can be specified in ASCII or hexadecimal and as regular expressions. Carving rules determine if the footer is included inside the recovered file, and if headers should be matched to the nearest footer, or the farthest away footer. Next, the first of two passes over the input file commences. The image is read in 10MiB chunks, each of which is searched for headers using a modified Boyer-Moore algorithm. If a header is found for a file type, Scalpel also searches for footers for that type. After all specified headers and footers are found, and the first pass is completed, headers are matched to footers according to the carving rules, and a schedule of carve operations is created. Using the schedule of carve operations, a second sequential pass over the image is made during which the candidate files are copied from the image into normal files in a user-specified directory.

Working with an already optimized tool emphasizes the effectiveness of the techniques discussed in the chapter. For example, Scalpel already performs sequential IO, unlike other file carvers, in that it performs two sequential passes over the input image. The first pass finds headers and footers which indicate candidate files. In the second pass, after headers and footers have been matched up, the data for each file is carved out of the image. It is important to note that in the second pass, regardless of the number of files carved, or whether or not they overlap, the data is read out of the image sequentially. This is an example of the first optimization technique discussed above. Before presenting any optimizations, we establish a performance baseline using the version of Scalpel discussed above.

The following experiments were performed on a Dell XPS 720 with a quad-core Core2 Extreme, 4GB of RAM, and 2 750GB 7200rpm SATA drives. This machine was equipped with an NVIDIA GTX260 GPU with 192 scalar processors and 896MB of RAM. All input and output was done on an 8 disk RAID-0 SAS attached Dell PowerVault MD1000, with a total of 1 TB of storage.

In order to enhance the repeatability and verifiability of these experiments, a much needed quality lacking in much of the current digital forensics research, the input disk image used is from the Digital Corpora, freely available at [45]. The specific image used is the *nps-2009-realistic.redacted.raw*, a 40G Windows XP SP3 image.

All timings are the average of three runs of the experiment (with reboots in between), for each of three sets of file types. We chose 3 runs and not more due to the relatively stable nature of a cleanly rebooted machine. Most of the very small variation between runs was due to background processing done by the operating system. The three file type sets represent three usage scenarios. The base case is a single file type, to demonstrate IO-bound performance when carving, and minimal output IO. The normal usage case is 15 file types commonly carved for in DFI. The last case is 60 file types, which is to demonstrate the CPU-bound case, and maximal output IO.

	Time	Speed	# Files	Space Required
1 Type	2m57s	231MiB/s	2,476	121MiB
15 Type	20m3s	34MiB/s	35,379	123GiB
60 Type	1h19m18s	8Mi**B**/s	141,516	494GiB

Table 5.2: Baseline Scalpel Numbers.

In table 5.2 we see that carving takes from 2m57s to 1h19m18s depending on the number if file types carved. For the 15 and 60 type cases, the size of the carved files is greater than the 40GiB size of the image they were carved from, at about 123GiB and 494GiB respectively. This is a function of the false positive rate, and necessitates significant amounts of temporary storage. The processing rates range from the respectable 231MiB/s when only searching for one file type to the positively sluggish 8MiB/s when searching for 60 file types. With targets on a terabyte scale, an 8MiB/s processing speed is clearly insufficient. In the next section, we begin the quest for efficiency by eliminating unnecessary IO using a technique termed *in-place* file carving.

5.5.1 In-Place Carving

The main use of file carving is to recover files which are no longer accessible in a filesystem in order to examine their contents, either visually or by using some other set of tools. In a sense, file carving is the process of giving recovered files, here byte streams, the abstraction offered by the filesystem. This is why the bytes of a file recovered from a byte stream are copied out of the raw stream and into a file. The abstraction offered by the filesystem gives any other tool the ability to access the contents of the file using a well-known API, including open(), close(), read() and write(). A significant inefficiency exists here. The bytes we are interested in recovering are already stored in the disk image. Copying them out of the image and directly into another file is a tremendous waste of IO bandwidth, storage, and the respective time it takes to write to the storage. In this section we present a technique termed *in-place* carving. This approach presents a filesystem interface to carved files without the performance penalty associated with actually carving out the bytes of recovered files.

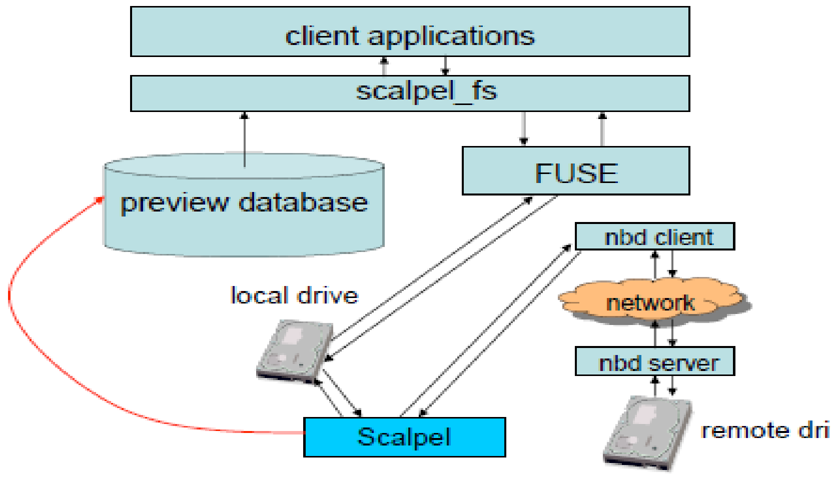


Figure 5.3: ***scalpel_fs*** **Architecture.**

In order to achieve the time and space savings characteristic of in-place carving, a multi-level system is employed. Our in-place carving architecture, *scalpel_fs*, is comprised of three major components: Scalpel v1.60, which provides a new "preview" mode, a custom FUSE filesystem (see section 3.3.3 for background on FUSE) for providing a standard filesystem view of carved files, and the Linux network block device (NBD) [146], which allows carving of remote disk targets. We support both live and dead investigative targets and both local and NBD disks. Similarly, carving operations may be performed either on the same machine that hosts the target disk device (or image) or on a separate investigative machine. The architecture is depicted in Figure 5.3..

Operating above a remote disk target is the network block device server, which provides live, remote access to the disk. Local disks are accessed directly by the Scalpel file carver, operating in a new "preview" mode. When operating in preview mode, Scalpel executes file carving rules specified in its configuration file to identify candidate files for carving on the disk devices. These files are only potentially interesting, as current carving strategies may generate copious amounts of false positives, depending on the carver configuration. Normally, Scalpel would then carve the disk blocks associated with candidate files and write them out to new files. In preview mode, however, Scalpel produces entries in a database detailing the starting position of the file on the device, whether or not the file was truncated, its length, and the device where it resides. The format is as follows:

```
filename          start       truncated   length      image
...
htm/00000076.htm  19628032    NO          239         /tmp/linux-image
jpg/00000069.jpg  36021248    NO          359022      /tmp/linux-image
htm/00000078.htm  59897292    NO          40          /tmp/linux-image
jpg/00000074.jpg  56271872    NO          16069       /tmp/linux-image
...
```

Figure 5.4: Scalpel Preview Mode Sample Output.

The original names of files are typically not available when file carving is used for data recovery, so Scalpel assigns a unique pathname to each carved file. The pathname indicates where the files would be created if Scalpel were not operating in preview mode. Our custom FUSE filesystem, *scalpel_fs*, accesses the Scalpel preview database and the targeted disk devices. The *scalpel_fs* application is implemented in C against the FUSE API and provides the standard Linux filesystem interface to files described in the Scalpel preview database, without carving the files from the target device. Executing *scalpel_fs* with arguments detailing a directory to be used as a mount point, a Scalpel preview database and the device that the database was generated from causes the following actions to be taken. First, a directory named *scalpel_fs* is created under the mount point. Then, using the carved file specifications in the preview database, a tree of filesystem objects is created in a structure determined by the filenames of the entries in the database. Files and directories are modeled as *fs_object* (see Figure 5.5) structures:

```
struct fs_object {
   int type; // file or directory
   char *name; // this object's fully qualified pathname
   int start; // starting index in the source image of the file
   int length; // size in bytes
   char clipped; // true if the file was truncated
   char *source; // the source image file name
   struct fs_object *children;      // empty for files
   struct fs_object *next;    // peer nodes
}
```

Figure 5.5: struct *fs_object.*

This tree structure provides the information necessary to present the user with a filesystem appearing to contain the carved files from the target devices. For efficiency, a pointer to each *fs_object* is also entered into a hash table for fast lookups. Listing the contents of the mounted directory shows one directory named *scalpel_fs*. Appearing inside the *scalpel_fs* directory are files and directories mirroring those in the filesystem tree created from the Scalpel preview database.

All file-oriented system calls targeting the mount point of the *scalpel_fs* filesystem are intercepted. Preceding most filesystem operations is a call to *getattr* for the filesystem object in question, which returns a *stat* structure containing information such as object type (file, directory, etc.), size, creation, access and modification times, and permissions. On receiving the *getattr* call, *scalpel_fs* dynamically constructs and returns a new *stat* structure with type and size taken from the *fs_object* for the object and creation / modification / access times and permissions duplicating those of the *scalpel_fs* directory. Directory listings are created by using the *children* structures of the *fs_object* for the directory being listed. Opening a file returns a file handle (after checking the existence of the *fs_object* in the hash table).

Attempts to read a file are handled as follows: the target device is opened and reading begins at the offset given by the *start* member of the *fs_object* structure for the file (plus any offset passed to the read operation itself). This is all transparent to the client application (and to the user). Other non-write filesystem operations also work transparently (e.g., *access, getattr, readdir*, etc.) Any operations that would create content (*write, mkdir, link* etc.) are disallowed in order to maintain the forensic soundness of the target. An exception is the delete (*unlink*) operation, which is allowed, but only in a shallow manner: the *fs_object* for the deleted file is removed but the target disk device is left untouched. This removes the file from the view provided by *scalpel_fs* without destroying any data.

At the top level of the system are other user-level applications. They can freely and transparently operate on the files under the special mount point as if they were regular files (aside from the disallowed write operations). A user can obtain cryptographic hashes of the files with hashing programs, view the files in text editors or image viewers, or use specialized forensics software on the files. This is particularly useful in an investigation involving image files (e.g., JPG or GIF images) as the images can be previewed as thumbnails by most filesystem browsers. Note that all of this occurs without the large amounts of space required by a normal carving operation.

	Time	**Speed**	**CFB**	**# Files**	**Space Required**
1 Type	2m47s	245MiB/s	1.06x	2,476	204KiB
15 Type	13m28s	51MiB/s	1.5x	35,379	2MiB
60 Type	49m17s	13MiB/s	1.63x	141,516	10MiB

CFB: change from baseline

Table 5.3: Scalpel With *in_place* Carving.

The performance benefits of this system are numerous. Table 5.3 shows the timing and output data size results for the in-place carving method. Note that as it should be, the number of files carved is identical. The gains in processing time are due to writing out metadata about the files recovered in place of the actual files themselves. As the number of files recovered increases, so do the gains in processing speed, due to the reduced IO requirements. This accounts for the 6% - 61% improvement in processing time. Much more interesting are the gains in required storage. For the 1-type case, storage requirements were reduced from 121MiB to just 204KiB. In the more extreme 60-type case the storage requirements shrank from 494GiB to 10MiB. This is approximately a 50,000 times reduction in the storage required. In order to reduce the time required by the remaining IO in the file carving operation, in the next section, we implement a form of non-blocking IO. Note that all further experiments in this chapter use *in-place* carving in addition to other performance enhancing techniques.

5.5.2 Non-Blocking IO

In the baseline Scalpel, 10MiB of the input image is read, and then searched. While the search is being conducted, no further attempts at reading more of the input image are made. Similarly, while input is being read from the image, no search is being conducted. Since modern machines are capable of DMA and can therefore transfer data from storage to RAM and back without the attention of the CPU, this read / process loop is inefficient. In order to be more efficient we implement a form of non-blocking IO such that reads from storage can be overlapped with search of the chunks read. This can be accomplished in several ways, but here we leverage the PThreads threading library for creating independently executing threads, and for the mutual exclusion constructs necessary for constructing thread safe queues.

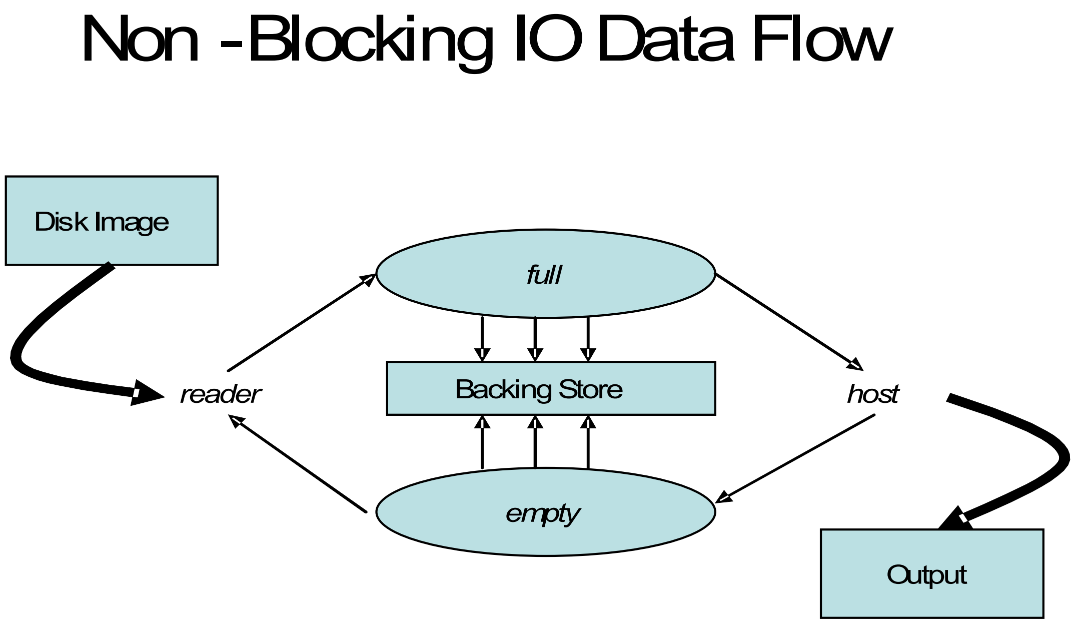


Figure 5.6: Scalpel Multicore Data Flow.

We first allocate a backing store of 10MiB buffers which will hold data read from the input image. We then set up two FIFO queues, *full* and *empty* (see Figure 5.6). Each queue is protected by a *pthread_mutex_t* to ensure its consistency when accessed by multiple threads. For each of the buffers in the backing store, a *readbuf* structure is initialized with a pointer to the buffer, and other accounting information and is place in the *empty* queue. Last, we initialize a *reader* pthread. When these setup steps are complete, the reader begins by getting a *readbuf* from the *empty* queue, reading a chunk of data from the input image into the buffer pointed to by that *readbuf*, and putting the *readbuf* into the *full* queue, and then getting another *readbuf* from the *empty* queue. Concurrent with this operation, the main host thread gets *readbuf*s from the *full* queue, performs the header and footer search on the buffer pointed to by that *readbuf*. When the search is finished, the readbuf is put into the *empty* queue, and another readbuf is acquired from the *full* queue.

This scheme increases the efficiency of Scalpel in several ways. The one-time allocation of the backing store eliminates the need to constantly allocate and free buffers during the program execution. The queues allow the reader thread to read input independently of the processing occurring in the main thread. Similarly, the main host thread can search buffers full of input data without necessarily having to wait for IO to complete. The practical efficiency however is clearly dictated by the specifics of the disks, images, machines etc. used on a particular program run. Results for Scalpel using non-blocking IO are shown in table <<>> below.

	Time	**Speed**	**CFB**
1 Type	1m41s	406MiB/s	1.76x
15 Types	11m59s	57MiB/s	1.68x
60 Types	47m21s	14MiB/s	1.75x

Table 5.4: Scalpel With *in_place* Carving and Non-Blocking IO.

The gains in processing speed with non-blocking IO are most pronounced for the 1-type case. Where in-place carving alone achieved 50% and 63% speed gains for the 15-type and 60-type cases respectively, it achieved a more modest 6% improvement in the 1-type case. Here the non-blocking IO freed the CPU to continue processing while IO was performed. This result is reasonable, as this the 1-type case is where we are most IO-bound since we are only searching for 1 file type. The techniques implemented in this section and the previous section have achieved significant gains by lessening the constraints imposed by inefficient IO usage. In the next section we begin to address the issue of processor resource utilization.

5.5.3 CPU Multithreading

In order to make efficient use of multiple processor cores, we instrument Scalpel for multi-threaded operation. During normal operation of our enhanced Scalpel, a single host thread acquires *readbuf* structures from the *full* queue and performs a modified Boyer-Moore string search for each of the file types it is configured to carve for, in sequence. Because this operation is the most computationally expensive work performed by Scalpel, and because it is an operation amenable to parallelization, this is where we implement the required changes. Instead of a single host thread searching for each file type in turn, we spawn a pool of threads at program initialization time, one for each file type Scalpel is configured to search for. As each block is grabbed from the full queue, these threads perform their searches in parallel. This thread pool uses two *pthread_mutex_t*s, *workavailable* and *workcomplete* as synchronization barriers to let threads know when to wait for more input data to search and to let the host know when all threads have finished with a buffer.

	Time	**Speed**	**CFB**
1 Type	1m45s	390MiB/s	1.69x
15 Types	5m34s	123MiB/s	3.62x
60 Types	13m33s	50MiB/s	6.25x

Table 5.5: Scalpel With *in_place* Carving and Non-Blocking IO and Multithreaded Search.

As seen in Table 5.5, utilizing multiple processor cores leads to a significant increase in the rate at which input can be processed. In the 1-type case, where the execution is IO-bound, we see little change from the previous set of experiments. As we search for more file types, and become more dependent on the processing speed of the system, the increase in the processing rate is much larger. In the 60-type case we process input more than five times faster than the baseline. In the next section, we implement the last of the efficiency techniques demonstrated in this chapter, by utilizing highly parallel co-processors.

5.5.4 GPGPU

In this section we first provide some background on GPU programming, based on NVIDIA's CUDA SDK, appropriate for use with NVIDIA's G80 and subsequent GPU architectures. This section is a summary of the information available in the CUDA SDK documentation at [123]; interested readers are referred to that document for more expansive coverage. The primary hurdles that a programmer unfamiliar with GPU programming will face are a complicated memory hierarchy, a Single Instruction Multiple Thread (SIMT) execution model (a relaxed version of the more common Single Instruction Multiple Data (SIMD) model), and the need to explicitly stage data in the GPU for processing. The programmer will also face familiarization with some other issues to obtain maximum performance. First, unlike multithreaded programming on traditional CPUs, where thread creation is expensive and therefore reusable thread pools are often used, GPUs create and destroy large numbers of threads with very low overhead. Furthermore, GPUs perform floating point arithmetic very quickly and memory accesses (particularly to the large, un-cached pool of device memory called global memory) are relatively expensive. This is likely to contradict the experience of programmers who have focused primarily on programming traditional computer systems. The remainder of this section discusses architectural details relevant to developers creating massively threaded tools for processing of digital evidence.

5.5.4.1 Execution Model

The latest NVIDIA GPUs are organized as a set of multiprocessors, each of which contains a set of scalar processors which operate on SIMT (Single Instruction Multiple Thread) programs. The GTX260 GPU, a commodity graphics card, and the model in the test system for these experiments, is typical, having 192 scalar processors organized into 24 multiprocessors with 8 scalar processors each. Each of these scalar processors executes at 1.242GHz, for a theoretical maximum compute capability of approximately 715 GFLOPs. A total of 896MB of RAM is available on the 260. Subsequent models, such as the GTX280, have more scalar processors, more RAM, and higher performance. Unlike earlier GPU designs, which had fixed numbers of special-purpose processors (e.g., vertex and fragment shaders), very limited support for arbitrary memory accesses (scatter/gather), and little or no support for integer data types, these scalar processors are general purpose. In addition to the set of scalar processors, each NVIDIA multiprocessor contains two units for computation of transcendental functions, a pool of shared memory, and a multithreaded instruction unit. Barrier synchronization and thread scheduling are implemented in hardware.

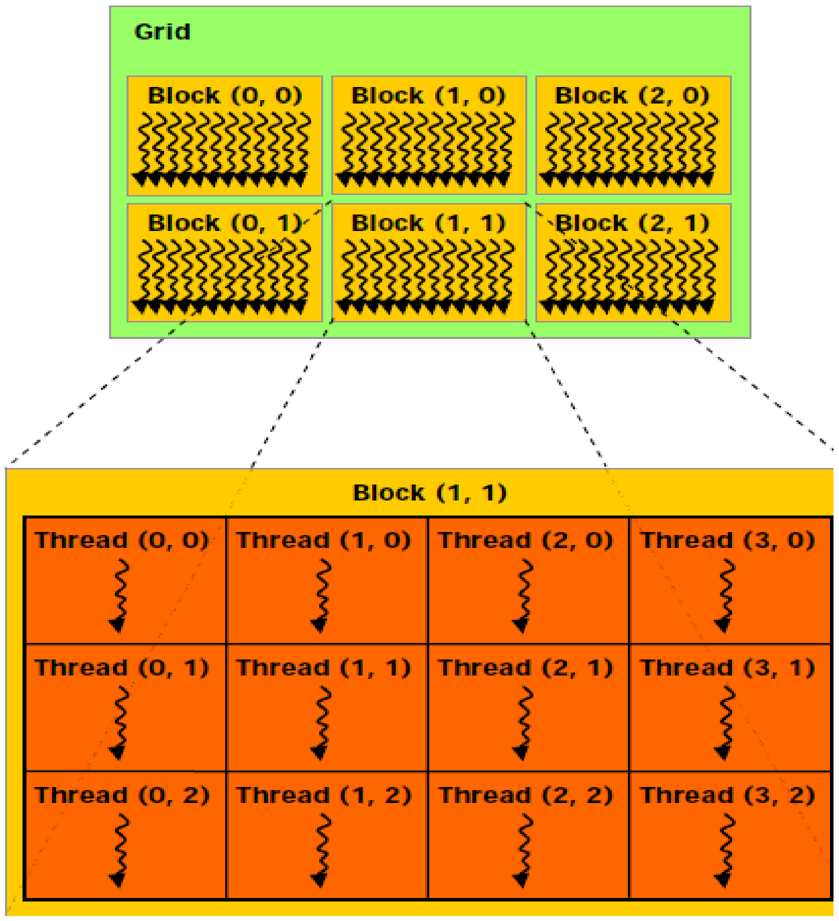


Figure 5.7: CUDA Thread Organization [123].

To utilize the GPU, the host computer is responsible for copying data to the GPU, issuing units of work to be performed, and then copying results back from the GPU once the units of work have completed. Each unit of work is called a *kernel* and defines the computation to be performed by a large number of threads. The highest level organization of the threads is a *grid*, with areas of the grid organized into *blocks* see Figure 5.7. Each multiprocessor executes the threads making up a block in parallel. Since the number of threads in a block may exceed the number of scalar processors in a single multiprocessor, the threads of a block are further organized into *warps*. A warp is a fraction of a thread block, comprised of a set of threads that are currently executing on a particular multiprocessor. As thread blocks complete execution, new thread blocks are launched on available multiprocessors.

The SIMT model deserves additional attention. SIMT is related to the more familiar SIMD execution model, but differs in some important ways. In NVIDIA's SIMT model, each scalar processor executes a single thread and maintains its own register state and current instruction address. A multiprocessor executes sets of threads called warps in parallel, with each thread able to execute instructions and perform branches independently. Since a multiprocessor has only one instruction unit, a warp executes one common instruction at a time. If control flow diverges between threads because of a conditional branch (e.g., taking opposite branches in an IF/THEN/ELSE), then the multiprocessor executes the divergent instruction streams serially until control flow resynchronizes. A key difference between SIMD and SIMT is that for the sake of correctness, programmers need not worry about control flow divergence. Only *efficiency*, not *correctness*, is impacted when threads execute independent code paths (resulting in serialization), which offers greater programming flexibility.

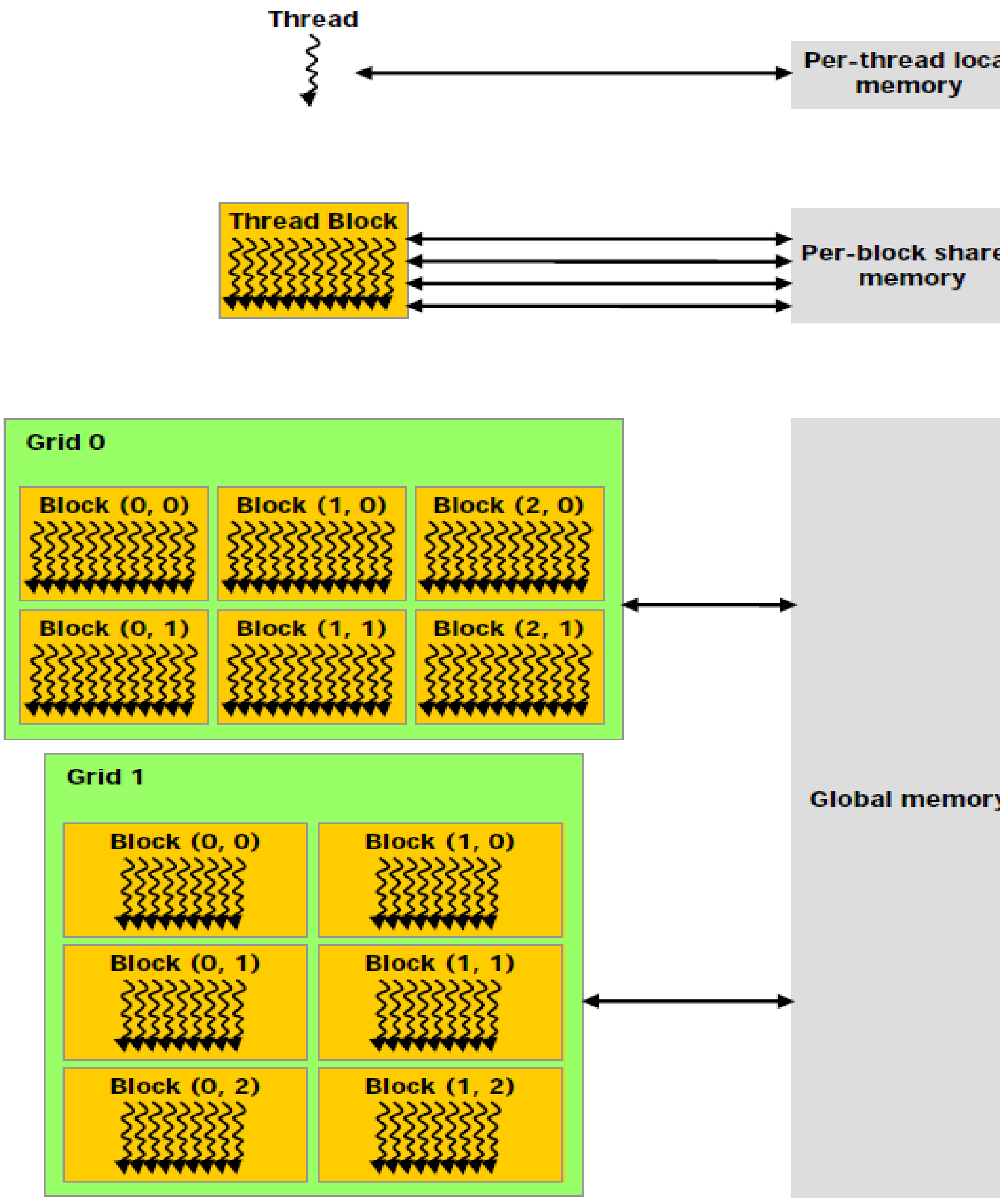


Figure 5.8: CUDA Memory Hierarchy [123].

5.5.4.2 Memory Architecture

NVIDIA GPUs provide a number of available memory spaces, through which threads can communicate with each other and with the host computer (see Figure 5.8). These memory areas, with restrictions and associated costs, are:

A private set of 32-bit registers, local to a particular thread and readable and writable only by that thread.

Shared memory can be read and written by threads executing within a particular thread group. The shared memory space is divided into distinct, equal-sized banks which can be accessed simultaneously. This memory is on-chip and can be accessed by threads within a warp as quickly as accessing registers, assuming there are no bank conflicts. Requests to different banks can be serviced in one clock cycle. Requests to a single bank are serialized, resulting in reduced memory bandwidth.

Constant memory is a global read-only memory space initialized by the host and readable by all threads in a kernel. Constant memory is cached and a read costs one memory read from device memory only on a cache miss, otherwise it costs one read from the constant cache. For all threads of a particular warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads.

Texture memory is a global, read-only memory space shared by all threads. Texture memory is cached and texture accesses cost one read from device memory only on texture cache misses. Texture memory is initialized by the host. Hardware texture units can apply various transformations at the point of texture memory access.

Global memory is un-cached device memory, readable and writeable by all threads in a kernel and by the host. Accesses to global memory are expensive, but programmers can use a set of guidelines discussed in the CUDA Programmer's Reference Manual to increase performance.

The most important issue governing these memory spaces is that access to the largest general purpose pool of memory, global memory, is flexible but expensive. Simply staging data that needs to be processed in global memory without attention to the other memory pools is easier, but will generally result in poor performance.

5.5.4.3 The Common Unified Device Architecture (CUDA) SDK

We now very briefly discuss the CUDA SDK, used for development of the tools described in the case studies in Section 4. CUDA is a compiler, set of development tools, and libraries for writing GPU-enabled applications. CUDA programs are written in C/C++, with CUDA-specific extensions, and are compiled using *nvcc* compiler (included in CUDA), under Microsoft Windows, Mac OS X, or Linux.

A CUDA program consists of a host component, executed on the CPU, and a device component, executed on the GPU. The host component issues bundles of work (kernels) to be performed by threads executing on the GPU. There are few restrictions on the host component. CUDA provides functions for managing the kernel invocations, memory management functions that allow allocating and initializing device memory, texture handling, and support for OpenGL and Direct3D. On the other hand, code that executes on the GPU has a number of constraints that are not imposed on host code. Some of these limitations are "absolute" and some simply reduce performance. In general, standard C library functions are not available in code executing on the GPU. CUDA does provide a limited set of functions for handling mathematical operations, vector processing, and texture and memory management. Recursion, static variables within functions, and functions with a variable number of arguments are not supported at all on the GPU component.

Finally, the *compute capability* of the GPU should be taken into account when using CUDA to develop GPU-enhanced tools. The compute capability of a GPU device is defined by a major revision number and a minor revision number. The initial series of NVIDIA GPUs that supported CUDA, including the G80, have compute capability 1.0, which provides basic features. Devices with compute capability 1.1 support all of the features of 1.0 devices plus the addition of atomic operations on 32 bit words in global memory. These operations include simple atomic arithmetic (e.g., *atomic_add()*), atomic swaps, min/max functions, etc. Compute capability 1.2 adds support for 64-bit atomic operations, adds support for "warp voting" functions, and doubles the number of available registers per multiprocessor. Warp voting functions allow the evaluation of a predicate across all currently executing threads on a multiprocessor and return a Boolean value. The latest compute capability as this chapter is written is 1.3, which provides support for double precision floating point numbers (devices with earlier computer capabilities support only single precision).

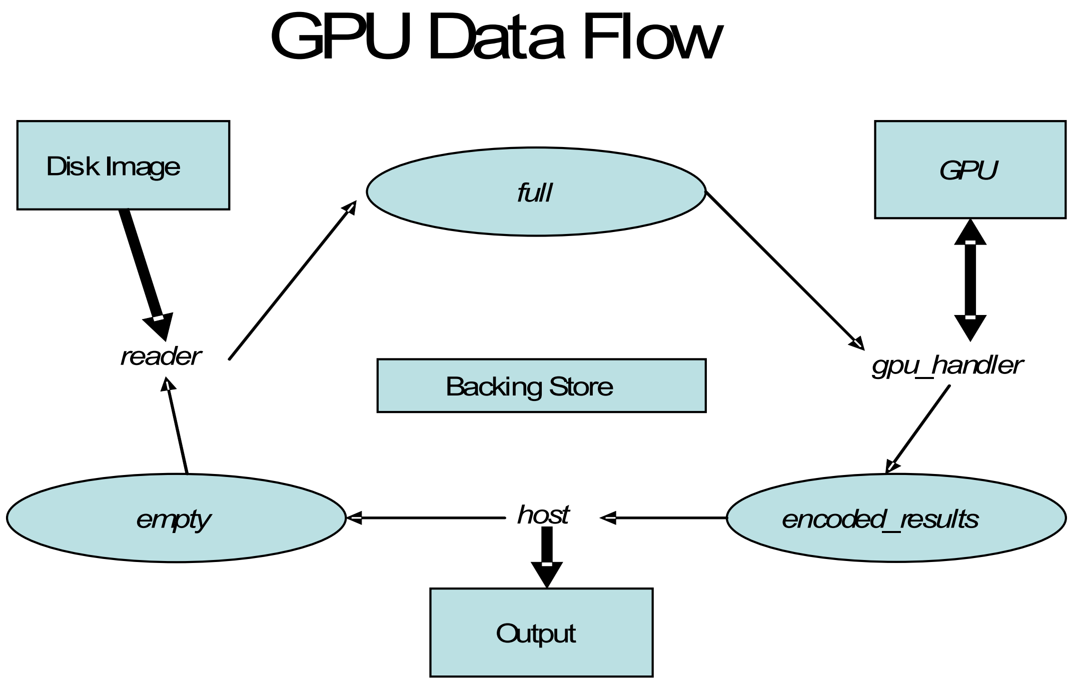


Figure 5.9: Scalpel GPU Data Flow.

In order to instrument Scalpel to utilize the CUDA-enabled NVIDIA N260 GPU in the test machine, several changes were made (see Figure 5.9). CUDA organizes sets of GPU operations (memory allocations, kernel executions) into *contexts* which are attached to individual host threads. Memory allocations on the GPU made in one host thread (in one context) are not visible to other host threads. For this purpose, a *gpu_handler* pthread was created to perform all GPU related operations. Because of bandwidth limitations between the host and device, the results of string searches are encoded on the device and then transferred to the host for decoding. To support concurrent IO, GPU binary string search, and CPU decoding of results, a third synchronizes queue was added to the system, in addition to *full* and *empty.* This third queue, *encoded_results* holds buffers of GPU encoded string search results.

After these changes, data flows through the system as follows. The *reader* thread acquires buffers from the *empty* queue, reads input data into the buffer, and puts the buffer into the *full* queue. The *gpu_handler* thread gets buffers from the *full* queue sends the contents of the buffer to the GPU for header and footer search, retrieves a buffer of results from the GPU and puts the buffer into the *encoded_results* queue. The main host thread gets buffers from the *encoded_results* queue, decodes the results and puts the buffer into the *empty* queue. See Figure 5.8. Note that moving buffers from queue to queue requires only the *readbuf* structure with the pointer to the buffer to be copied, not the actual 10MiB backing buffer.

The GPU requires several initialization steps before it can be used efficiently, starting with memory allocations. A buffer of memory in the global GPU memory space is allocated for input data from the host, *dev_in*, as well as one for holding encoded results which will be sent to the host after string search completes, *dev_out*. These allocations persist for the entire duration of the program execution, and are re-used every time input data is moved to the device, or results are retrieved from it. We then perform initializations related to the string search algorithm used by the GPU. First we create a table of all of the strings the GPU is to search for, the headers and footers for each file type, named *patterns*. Next, in the interest of efficient string search, we create two lookup tables, *lookup_headers* and *lookup_footers* constructed as follows. Each table contains 256 rows, indexed 0 to 255. Each row contains a list of indexes into the *patterns* table for search strings which begin with the index of this row. For example, *lookup_headers*[7] holds a list of indexes into the *patterns* table of headers which begin with the byte \x07. If there are three such headers, the eighth row in *lookup_headers* will contain the indexes in the *patterns* table for those three headers. After being constructed, these three tables are copied to the GPU, into space allocated from the constant memory pool. Since there is a local cache for each multiprocessor for constant memory, multiple threads accessing the tables gets faster over time, as more of the table is cached locally.

With these initialization steps taken care of, we discuss the binary string search implementation on the device. The *gpu_handler* gets 10MiB chunks of the input image from the *full* queue and copies the data to the GPU's *dev_in* buffer. Then a GPU kernel is launched to perform the actual search. A number of 256-thread blocks are initialized equal to the size of the input chunk. Each thread block copies 256 bytes of the input from *dev_in* to a local buffer from the shared memory pool. This staging area enables fast access to the data, and is accessible only to this thread block. Each of the 256 threads in the thread block then assumes responsibility for one of the 256 starting byte positions in the staging area. Each thread then examines the byte value stored at its starting position, and looks it up in each of the tables, *lookup_headers* and *lookup_footers.* If the tables point to search strings in the *patterns* table, a simple search is performed to see if those headers or footers exists starting at this position in the staging area. Any matches are encoded in the *dev_out* buffer. When all thread blocks have finished searching and recording results, the *gpu_handler* copies the *dev_out* buffer off of the GPU and puts it in the *encoded_results* queue for decoding by the main host thread.

The result of using the GPU for string search as opposed to the multiple CPU cores is even more significant improvements on processing speed, as seen in Table 5.6 below.

	Time	Speed	CFB
1 Type	1m48s	379MiB/s	1.64x
15 Types	2m45s	248MiB/s	7.29x
60 Types	6m44s	101MiB/s	12.63x

Table 5.6: Scalpel With *in_place* Carving and Non-Blocking IO and GPU-based Search.

Aside from the 1-type case where we are presumably completely IO-bound, the GPU is almost exactly twice as fast as the multicore implementation. This is impressive for a host of reasons. First, these improvements are in spite of comparison to a highly optimized multicore implementation running on a machine with 4 very fast cores..And, there is still room for improvement as there is still potential for optimization of the GPU code. Further, string search is not a tremendously computationally expensive operation. If applied to operations which exhibit more arithmetic intensity, the GPU could provide even greater gains.

5.6 Discussion

The focus of this chapter is improving the efficiency of digital forensic tools. As the number of cases requiring investigation, and their size, continues to increase, we require efficient tools to process the data within the given time constraints. The current generation of digital forensics tools performs poorly in terms of efficiency, with few making use of multicore processors or massively parallel co-processors, maximizing IO throughput, or conserving storage space. To help alleviate these problems, we presented a set of efficiency techniques and applied them to the Scalpel file carver. First, we used the *in-place* file carving technique to significantly reduce the total IO done by Scalpel, and its storage requirements. We then implemented a non-blocking IO scheme to maximize the use of IO bandwidth. Next we modified Scalpel's binary string search algorithm to take advantage of multicore processors. Last, we leveraged the massively parallel NVIDIA GTX260 GPU to increase processing even more. In all, the application of the efficiency techniques discussed here have garnered improvements over the baseline for in-place, non-blocking IO, GPU search based carving of greater than 12 times the processing speed, and potentially much larger improvements in disk storage usage. The application of GPUGPU techniques was the first in the digital forensics community. Because of the potential for faster processing exhibited in this work, similar GPU-based techniques have been applied to such problems as MD5 hashing and password cracking. There results achieved here are a clear indication that many other digital forensics tools could see tremendous benefits from the application of these techniques. Now that we have discussed improving the reliability, comprehensiveness and efficiency of DFI, we move on in the next chapter to constructing a coherent picture of the events in question in an investigation, given the copious amounts of data at our disposal.

.

Chapter 6:
Coherence

In previous chapters we have discussed methods for increasing the effectiveness of digital forensic investigation (DFI) in terms of reliability, comprehensiveness, and efficiency. Once an investigator has generated data from reliable and efficient tools with comprehensive coverage of the types of data on the digital devices available for investigation, a coherent picture of the sequence of events and state of the systems under investigation must be constructed. From the copious amounts of data generated by a plethora of DFI tools and techniques, creation of a single integrated view of events which answers the questions under investigation is the next step in the investigative process.

6.1 Current Practice

Currently, most DFIs are conducted using one of a handful of graphical tool suites, such as FTK, Encase, Sleuthkit / Autopsy, or PyFlag. Each of these offers some common functionality for parsing multiple filesystems, computing file hashes, creating full-text indexes and generally organizing data to make it useful to an investigator. Each also offers additional functionality not necessarily common to all of the others. For example, PyFlag has a configurable facility for parsing log files, and Autopsy can construct filesystem timelines.

To analyze types of data not handled by the suites above, (for functionality outside the scope of that offered by the above tools,) the investigator must use additional tools from a large (and growing) set of single purpose tools. These may have graphical or text-based interfaces, be free and open source or commercial and expensive. The process followed is rather ad-hoc and revolves around analyzing some of the data output by a tool, discovering some useful bit of information, and using the new information, pursuing some new avenue in order to discover more evidence (following “leads”). Any investigative work done outside the confines of a single tool suite forces the investigator to manage the data generated manually. This situation influences many investigators to restrict their investigations to the functionality provided by one of the major tool suites.

It is up to the investigator to manually aggregate the data generated by whichever tools are used into a coherent model of the events which took place. This is a difficult task for many reasons, some of which are the subject of the next section.

6.2 Issues

There is a veritable laundry list of difficulties in constructing a coherent view of the events which have transpired on a digital system.

First, the tool suites discussed above are all restricted to a basic set of functionality that is generally insufficient for conducting a comprehensive investigation. Further, they tend to be updated with new functionality significantly more slowly than new techniques are made available. As new technologies require even newer tools for their analysis, the gap between what is provided and what is needed grows wider. Further, tool suites,and single function graphical tools can make exporting data, so that it can be combined with other data, difficult or impossible.

If an investigator uses a set of single purpose tools in order to be free of the restrictions imposed by a tool suite, a whole other set of problems arises. In the case of text-based tools, each has its own command line syntax and set of options. Becoming competent with any large set is a non-trivial task. Assuming that there is any documentation for each of them, the quality of the documentation can range from precise and detailed to very poorly written and misleading. Even the terminology used can have different meanings across tools, making deciphering their exact function difficult. Similarly, for units of measurement, as some tools use base-1000 and others use base-1024 for measuring storage. Many tools perform similar functions, but return different results due to the quality of their respective implementations. This forces a thorough investigator to use multiple tools and aggregate their outputs. On the subject of output formats, there is no widely recognized standard. Some use XML, and some HTML, but most use simply structured text. Aggregating data across several tools, whether of similar or disparate function must be done manually. This requires strategies for parsing individual tool outputs, and ad-hoc record keeping.

All of the above problems underlie a larger one. In a DFI, the investigator must be able to make sense of the vast amount of information which a set of tools provides. These tools operate at various levels of a computer system, some procuring data from raw disk bytes, some at the level of filesystem metadata, and others at the level of application files, to name a few. They also operate on various types of media, such as network packet captures, copies of physical RAM, or myriad types of disk devices. Each tool is non-interoperable with any other, in terms of usage, terminology, and output format. This leaves the investigator the arduous and time-consuming task of mentally piecing together a picture of the state of the system under investigation; assembling this multi-dimensional puzzle requires tremendous cognitive effort. It is also work that should be assisted by computational tools.

In the next section we present a technique to aid the investigator by automating the process of building a coherent view of system state from the outputs of various unrelated forensic tools. Specifically, we present the Forensics Automated Coherence Engine (FACE), a data fusion framework for DFI. We define *data fusion* as combining data from multiple heterogeneous sources into a consistent, accurate composite picture of a system.

6.3 Proposed Solution

The impetus for conducting DFIs is to answer a set of questions about events that occurred in the past. These questions are generally, “did some specific event occur?”, and if so, “when did it occur?”, “how was it performed?”, and “by whom?” In an attempt to answer these questions an investigator will leverage the tools and techniques at his disposal to generate evidence which supports a coherent theory of events. The answers provided deal with people, possession and transmission of sensitive data, times when events occurred, and actions which set these events in motion. There is a semantic gap here. An individual tool may provide information about some small piece of data on disk in the system, or show that a web page exists in the browser cache. The leap from this relatively low-level data to answering the questions posed in the investigation is left to the investigator to perform with little or no help from the computational resources at his disposal.

FACE helps bridge this gap. It is designed to present data generated by DFI tools in a format which helps to answer the questions under investigation. To accomplish this, we start by identifying some high-level objects of interest in the system – users, running processes, files, network connections, and events. Most forensics tools give windows into specific facets of these objects, as when a browser cache parsing tool gives information specifically on the web browsing history located in a user home directory, or when a wtmp file parser gives information on user login history. Each of these examples gives some data about the same type of object in the system, namely user objects, and it is the focus of FACE to aggregate this data and present it as such. The list of these linkages between low-level data and higher-level objects is vast, and, for the moment, all the "linking" must be done in the investigator's head. Automating this process will go a long way towards enabling more effective DFI. To accomplish this, FACE was designed with the following goals:

- *High-Level System View.* FACE presents a view of the system in terms of the high-level entities on the system, specifically: Users, Processes, Files, Network Connections and Events.
- *Integrated View of Entities.* The interface presents all known information about the high-level listed entities above regardless of the source of the data, collected in a concise view.
- *Extensible.* The plugin API facilitates the inclusion of data from built-in tools, other currently available tools, and tools to be developed.

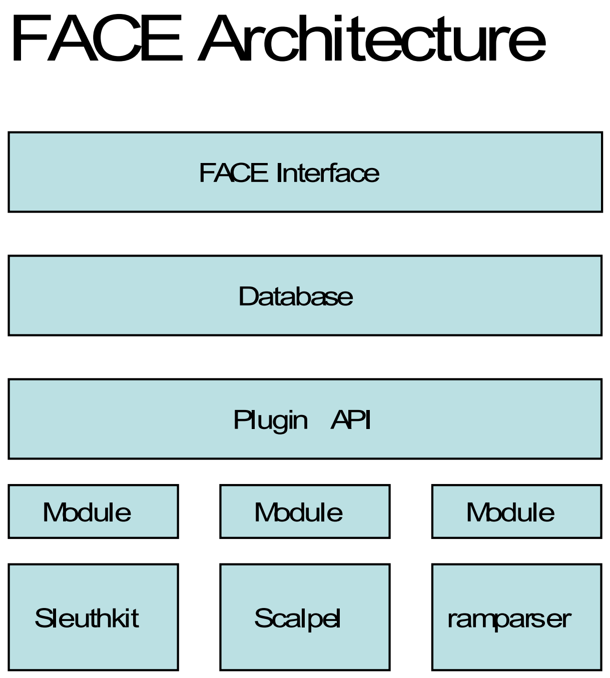


Figure 6.1: FACE Architecture.

6.3.1 Architecture

FACE is a data fusion framework for DFI (see Figure 6.1). It consists of a web-based front end, on top of a MySQL database, with support from the Django [147] web framework. Below the database is the pluggable API layer, and below that is a set of modules using the API. A base set of modules is included with the system. At the bottom of the stack is the raw data level; this is the data generated by DFI tools which are to be included in the framework. The web-based front end allows for investigations to be performed over the network by many clients, while the "heavy lifting" can be done on a single more powerful machine. The MySQL database is perfectly suited to generating responses to investigator-submitted queries against the data in the framework. The API, which is discussed in more detail below, allows new types of data to be easily imported into the system. Modules written using the API allow users to add new types of data into the system.

```
class User(models.Model):
      user_name = models.CharField(max_length=32, editable=False)
      password = models.CharField(max_length=32, editable=False)
      uid = models.PositiveIntegerField(editable=False, primary_key=True)
      gid = models.PositiveIntegerField(editable=False)
      comment = models.CharField(max_length=256, editable=False)
      home_dir = models.CharField(max_length=256, editable=False)
      shell = models.CharField(max_length=256, editable=False)
      primary_group = models.ForeignKey('Group')

      class Meta:
            ordering = ['uid']

      def __str__(self):
            return '%i %s %s %i %s %s %s' % (self.uid, self.user_name,
      self.password,\
            self.gid, self.comment, self.home_dir, self.shell)
…
```

Figure 6.2: Module for Top-Level Object: User.

6.3.2 Plugin API

The plugin API supports the creation of modules which facilitate the inclusion of raw DFI tool output into the FACE framework (see Figure 6.2). Modules are written in the Python programming language, which offers the ability to call out to native C code for efficiency when dealing with computationally expensive operations. Each module specifies mappings from raw data to Python objects. For each type of object specified by a module, several pieces of information are required. The name, and data type of each attribute of the object must be specified, as well as which attribute or set of attributes uniquely identify an instance of the object. To support the data integration functionality, the module must also specify which top-level object(s) this object is related to. Last, the module may specify a display function; if one if not specified, a default is used.

Modules can be created for any type of tool, based on any type of raw data. Using the data and linkages provided by modules, we can get a more complete view of the system. Such a view is possible if all available information is considered, including the memory dump, filesystem, log files, and network traces. The memory dump allows for reconstruction of all processes that were running on the system, and for per–process analysis, such as collecting open files, active sockets, and memory mappings. The packet capture contains timestamps and streams which can be used to easily match network traces to active sockets on the system. The wtmp and utmp files store information about when users logged in, their login location, and the time of login. Timelines and the correct order of events are critical during investigations, and by using the timestamps from file MAC times, login files and network traces the engine can accurately frame what actions a user performed during specific times. The passwd and group files map users to their user id, group id, home directory, and login shell. On–disk metadata and kernel structures only refer to user ids and group ids which are not friendly to a user, but by incorporating information from the passwd and group files, the UI can present the user with the names representing these groups and users. This also speeds up investigations since a user can be quickly identified based on their username.

A fairly complete reconstruction of the state of the running machine and its network activity becomes possible by fully parsing and analyzing these sources. By integrating the information, we are able to link data in a network trace back to the user who started the process which caused the traffic. This is possible by first matching an open socket in the memory dump to packets in the network trace. Since we know what process owns the open socket, we can then determine what user started the process and from where they logged in. It is also possible to determine what file on disk was being transferred by observing the open files of the process. Our system can also validate data currently queued in the kernel's network stack with data of the network trace. Partially transferred files can be fully reconstructed by joining these two sets of data together.

Analysis of malicious or unknown binaries or processes is made easier as the framework allows the user to download all or part of a process' memory or a file's data. Similarly, sections of memory can have more traditional forensics procedures applied to them such as file carving and hashing. Our framework also allows an investigator to get categorized views of disk and user activity at the time of the memory dump. Using FACE, it is possible to display all activity from a single user such as open files, active network connections, and running processes.
The rest of this section describes the various components in our framework, including parsers for network traces, configuration and log files, and the integration engine.

6.3.3 Standard Modules

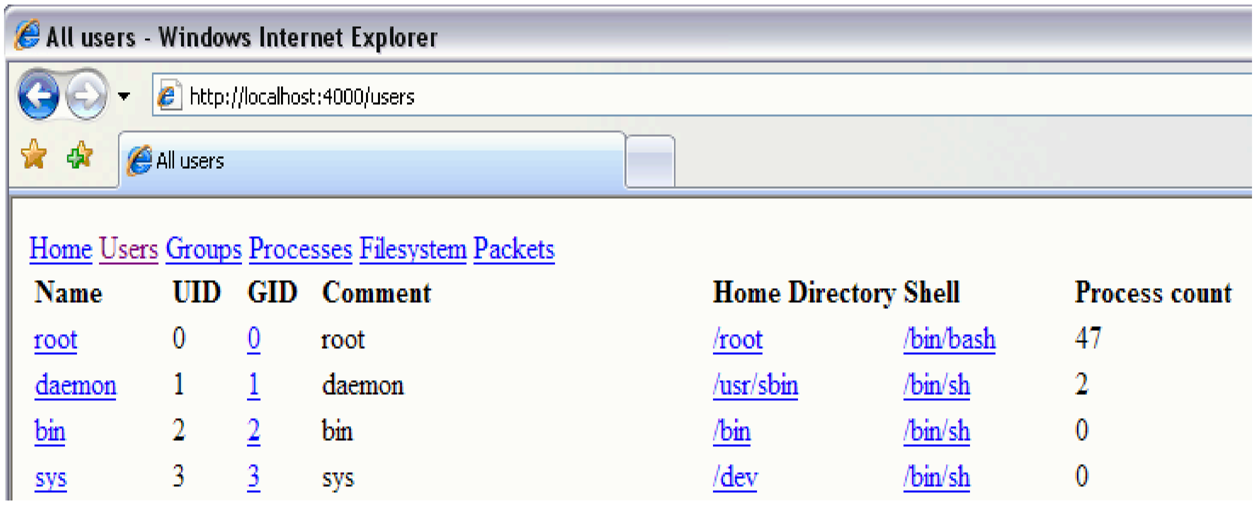
In addition to a module for *ramparser*, described in chapter 4, we implemented additional modules, discussed below.

Pcap Parser: To avoid duplicating the effort of excellent network capture parsing tools like Wireshark, we opted to implement a much simpler module for parsing captures into a format for the integration engine to work with. Our module reads in a pcap-format capture file and breaks it down into streams. Here we define streams as a collection of packets having the same source and destination IP address and port. This allows the integration engine to display inflows (collections of packets originating from the host) and outflows (those packets originating from elsewhere but destined for the host). For each of these streams the module outputs the type: TCP or UDP (other protocols are not currently implemented), and a list of the packets in the stream. For each packet in the list, we output the timestamp from the pcap header for the packet and the beginning and ending offsets of the packet in the capture file. Output is to a plain text file in a LISP-like s-expression format.

Configuration / Log File Parsers: In order to glean more information from the target system, we wrote three simple modules to parse a selection of files from the target filesystem. First, we parse /etc/passwd for information about users on the system (home directory, default shell, the contents of the comment field, etc) and user id to username mappings. The next module parses /etc/group for group membership information and group id to group name mappings. The third module parses /var/log/wtmp. This binary format file contains information on user logins to the system - namely the username, time and date of login, and where the login was from (local or some remote host). Each of these modules outputs a text file in a format similar to the one used for the pcap parser. Note that the files chosen are only a small subset of the forensically interesting files in the filesystem. In the future we intend to implement similar modules for many other interesting files (e.g., /var/log/messages, configuration files for servers, /etc/fstab, and /etc/xinetd.conf).

6.3.4 Framework

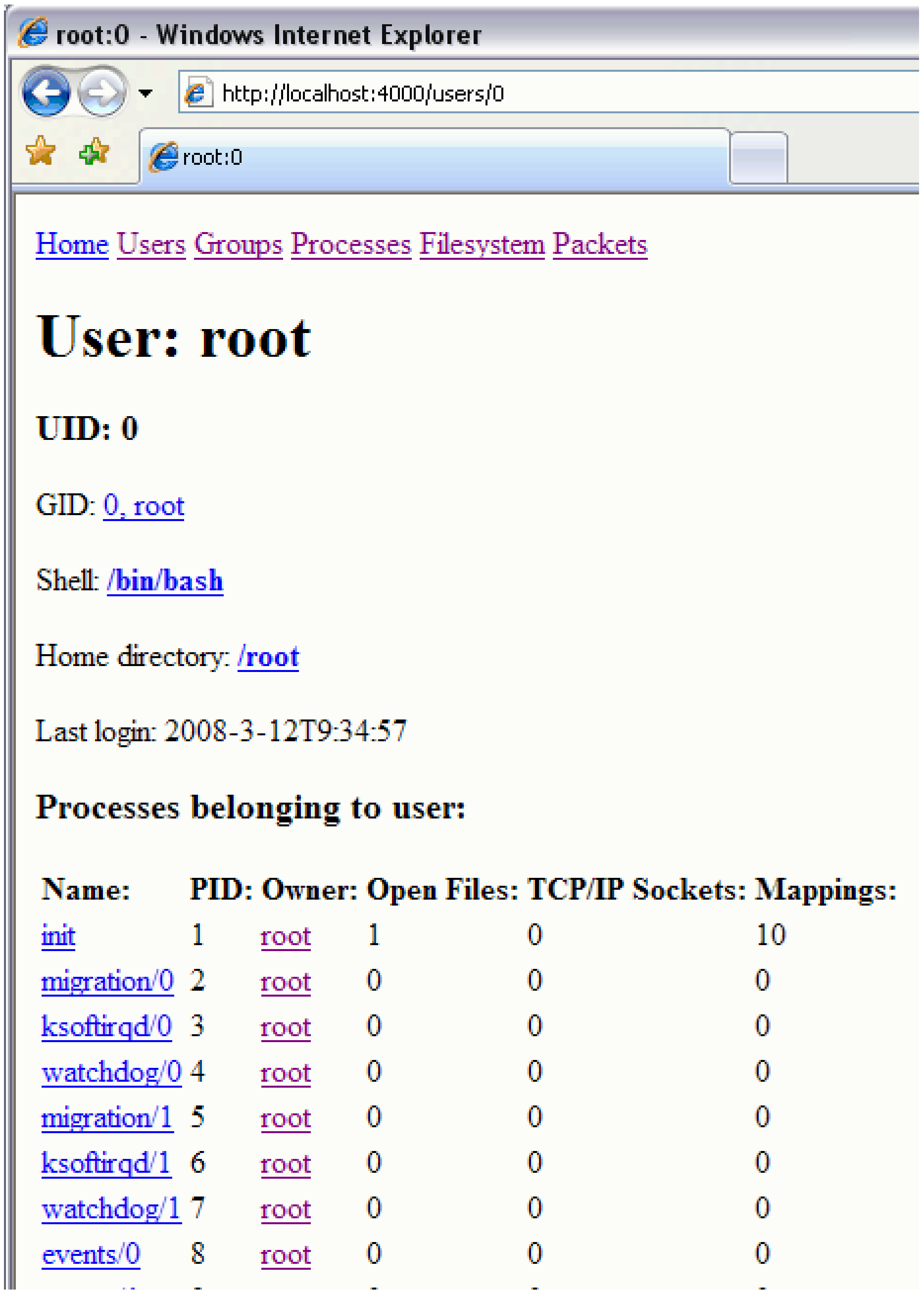
The overall vision behind FACE is to be an extensible platform for integrating and visualizing the logical connections among objects and events discovered by various evidence collection tools. Clearly, both the integration and visualization aspects are open-ended problems and our goal is to build the essential framework around which ever more sophisticated integration and visualization components can be attached. This approach will also open up the opportunity to adapt established solutions from other areas to the forensics domain. In its first incarnation, FACE is web-based and pulls together results from a number of standalone forensics and system administration utilities to present a hyperlinked interactive interface. The tool is written in Python, and uses a MySQL database to store and relate the various conceptual units (such as user identities, filenames, etc.) together. Currently, the framework has modules to integrate the output from, *ramparser*, most Linux filesystems, the login log file (wtmp), the /etc/passwd file, the /etc/group file, and pre-processed pcap files. Adding support for new sources of data involves writing a module using the provided API. Activating the new additional capabilities is done by updating the tool's configuration file; no further software integration effort is necessary.


All users - Windows Internet Explorer
http://localhost:4000/users
All users
Home Users Groups Processes Filesystem Packets

Name	UID	GID	Comment	Home Directory	Shell	Process count
root	0	0	root	/root	/bin/bash	47
daemon	1	1	daemon	/usr/sbin	/bin/sh	2
bin	2	2	bin	/bin	/bin/sh	0
sys	3	3	sys	/dev	/bin/sh	0

Figure 6.3: Users View.

The current version of FACE presents users with five main data views: users, groups, processes, filesystems, and network captures. Each of these main views can display a listing of all entries in that category, or a particular entry in more detail. Most fields in the detailed view are displayed as hyperlinks to related information (possibly viewed in a different tool). The user list (Figure 6.3) displays the user's name, their UID, GID (linked to the group entry), the comment field from /etc/passwd, their home directory and shell (linked to their entries in the filesystem), and a count of currently running processes for that user.


root:0 - Windows Internet Explorer
http://localhost:4000/users/0
root:0
Home Users Groups Processes Filesystem Packets

User: root

UID: 0

GID: 0, root

Shell: /bin/bash

Home directory: /root

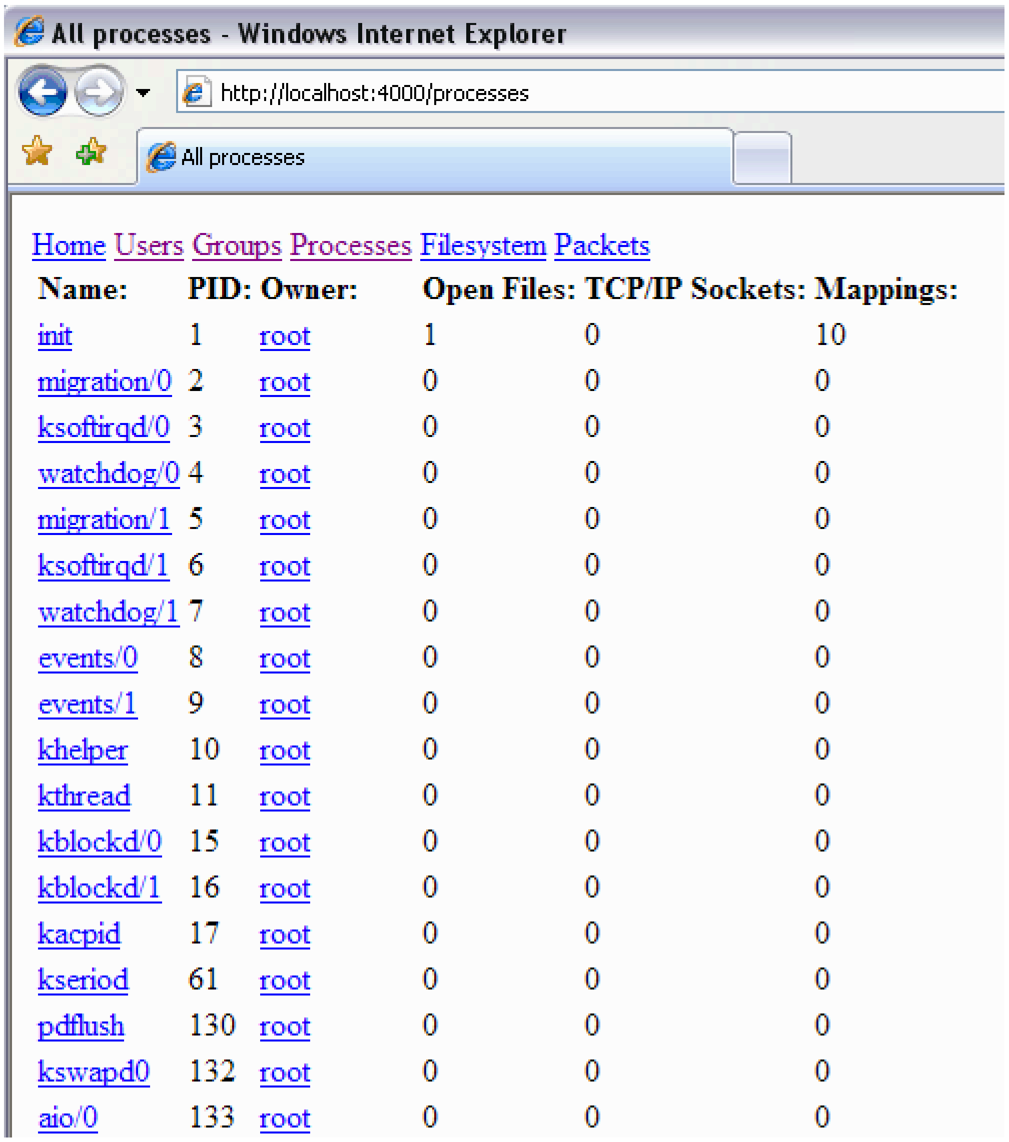
Last login: 2008-3-12T9:34:57

Processes belonging to user:

Name:	PID:	Owner:	Open Files:	TCP/IP Sockets:	Mappings:
init	1	root	1	0	10
migration/0	2	root	0	0	0
ksoftirqd/0	3	root	0	0	0
watchdog/0	4	root	0	0	0
migration/1	5	root	0	0	0
ksoftirqd/1	6	root	0	0	0
watchdog/1	7	root	0	0	0
events/0	8	root	0	0	0

Figure 6.4: Detailed User View.

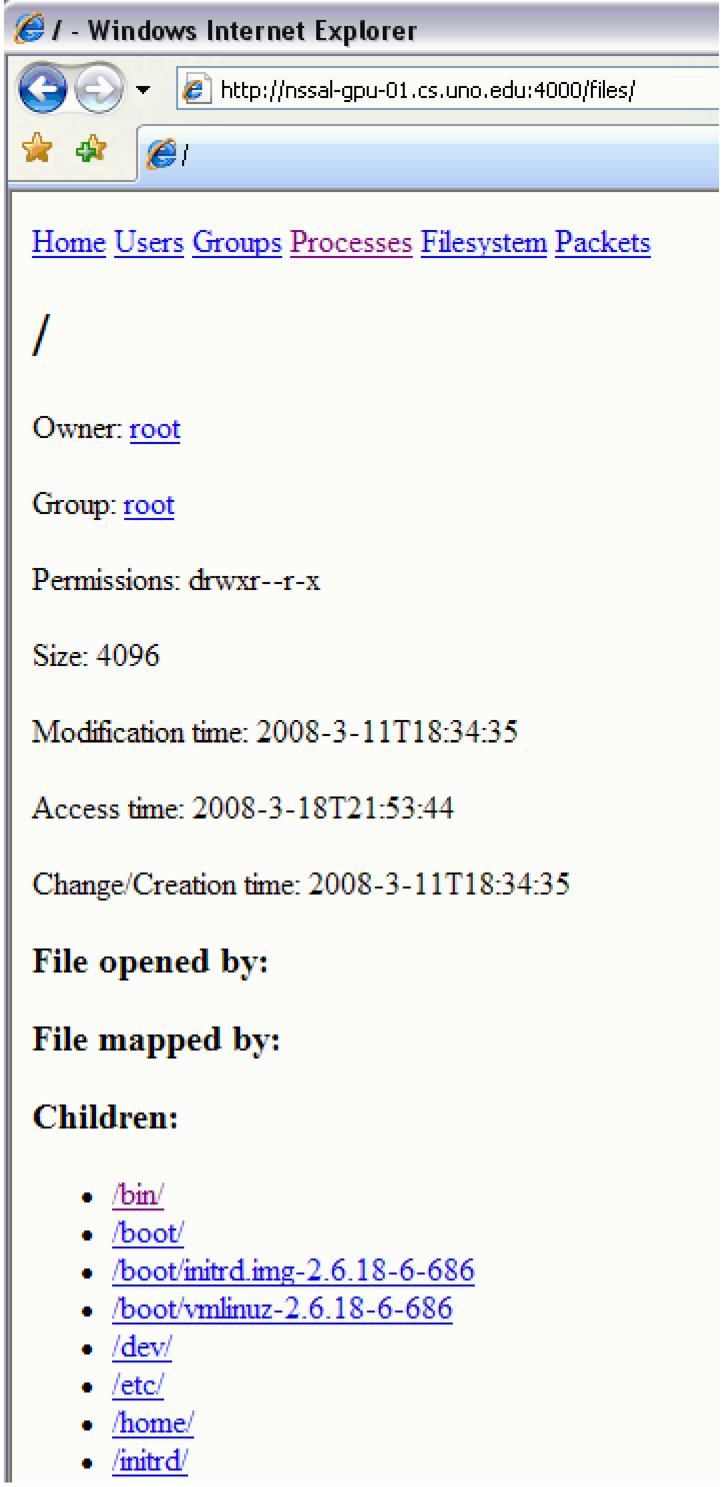
In the detailed user view (Figure 6.4), some additional information is presented, such as last login time, and a listing of all processes run by that user, and a list of all files currently opened by that user. This view is composed of information obtained by integrating data from *ramparser*, wtmp, /etc/passwd, and /etc/group. The files and processes are linked to their respective detail screens. The groups view displays a listing of all groups, detailing the name of the group (as a link to the group's detailed entry), the GID of the group, and the primary and supplementary members of the group (as links to those member's detailed user views).


All processes - Windows Internet Explorer
http://localhost:4000/processes
All processes
Home Users Groups Processes Filesystem Packets

Name:	PID:	Owner:	Open Files:	TCP/IP Sockets:	Mappings:
init	1	root	1	0	10
migration/0	2	root	0	0	0
ksoftirqd/0	3	root	0	0	0
watchdog/0	4	root	0	0	0
migration/1	5	root	0	0	0
ksoftirqd/1	6	root	0	0	0
watchdog/1	7	root	0	0	0
events/0	8	root	0	0	0
events/1	9	root	0	0	0
khelper	10	root	0	0	0
kthread	11	root	0	0	0
kblockd/0	15	root	0	0	0
kblockd/1	16	root	0	0	0
kacpid	17	root	0	0	0
kseriod	61	root	0	0	0
pdflush	130	root	0	0	0
kswapd0	132	root	0	0	0
aio/0	133	root	0	0	0

Figure 6.5: Processes View.

The process listing (Figure 6.5) shows every process running on the system (as per *ramparser* memory analysis), giving the name, the PID of the process, the user running the process, and a count of open files, open TCP/IP sockets, and memory mappings. The detailed process view also allows the investigator to view the code segment, data segment, stack or heap of the process in a hex dump like format or as raw bytes, suitable for saving to a file for further analysis. This detailed view also shows opened files and their file descriptor ids. If the open file is a real file, it is displayed as a link to the filesystem view of that file. If it is a TCP/IP socket, it is a link to the display page of that socket. The next piece of displayed data is a listing of all TCP/IP sockets for this process. Each entry gives the inode of the socket, the source IP:port and destination IP:port pair, and a count of the number of entries in the send and receive buffers. Clicking on the inode number will present the user with a detailed view of that socket. Finally, the detailed process view shows memory mappings for the process. Mappings of actual files, opened by mmap, or as a shared library, will be displayed as links to the filesystem view of those files. In addition, hexadecimal or raw displays of code and data segments for libraries or data for regular files are available. Similarly, anonymous areas of mapped memory are identified as such and can be viewed.



/ - Windows Internet Explorer

http://nssal-gpu-01.cs.uno.edu:4000/files/

Home Users Groups Processes Filesystem Packets

/

Owner: root

Group: root

Permissions: drwxr--r-x

Size: 4096

Modification time: 2008-3-11T18:34:35

Access time: 2008-3-18T21:53:44

Change/Creation time: 2008-3-11T18:34:35

File opened by:

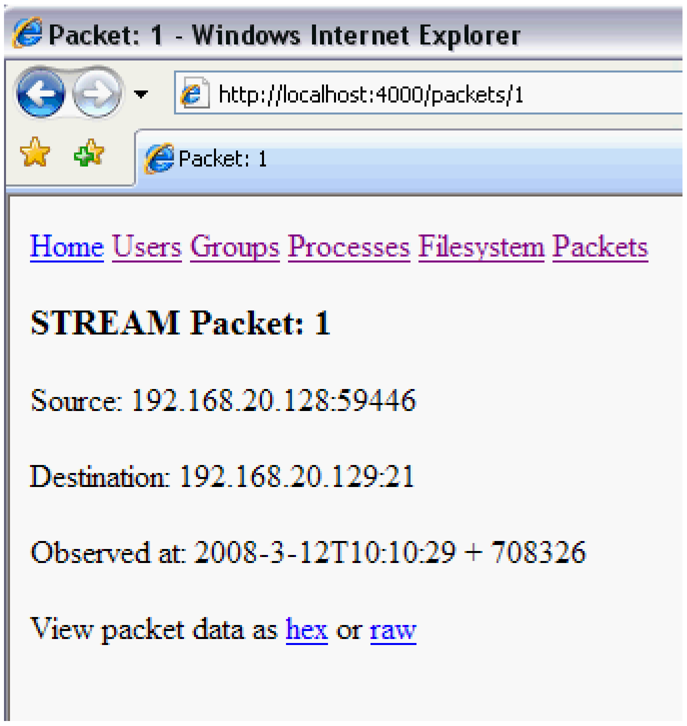
File mapped by:

Children:

- /bin/
- /boot/
- /boot/initrd.img-2.6.18-6-686
- /boot/vmlinuz-2.6.18-6-686
- /dev/
- /etc/
- /home/
- /initrd/

Figure 6.6: Filesystem View.

The filesystem view allows the investigator to browse a filesystem associated with the investigation. Each entry in the filesystem view is addressed by the URI /files/<path to file>, with /files/ representing the root directory. An entry in the filesystem view is treated as either a directory, or a regular file. In either case, the entry will display the owner's username and group (with appropriate links), the permissions flags, the size, the modification, access and change (MAC) times, as well as a listing of all processes which currently have the file opened or mapped. If the file is a directory (Figure 6.6), the entry will also list all directory entries as links to their specific filesystem pages. If the file is anything other than a directory, the name of the file will be displayed as a link to the actual file as it exists on the filesystem. The investigator may use this link to save a copy of the file for further analysis.


Packet: 1 - Windows Internet Explorer
http://localhost:4000/packets/1
Packet: 1
Home Users Groups Processes Filesystem Packets

STREAM Packet: 1

Source: 192.168.20.128:59446

Destination: 192.168.20.129:21

Observed at: 2008-3-12T10:10:29 + 708326

View packet data as hex or raw

Figure 6.7: Packet View.

The main packet view simply lists all packets found in the pcap network trace. The packets are assigned a unique id, and displayed as links to the individual packet's detailed view. The detailed view of a single packet (Figure 6.7) lists the source IP and port, the destination IP and port, the time at which the packet was captured, and allows the investigator to view the packet in both hexadecimal view and as raw bytes.

Clicking on a socket's inode will present the investigator with a detailed view of that socket. This view gives the bound port and IP (if any), the connected peer's IP and port (if any), and will give a listing of all incoming and outgoing packets associated with this socket. The investigator may view packets individually, or they may view the entire stream in hex or as raw bytes.

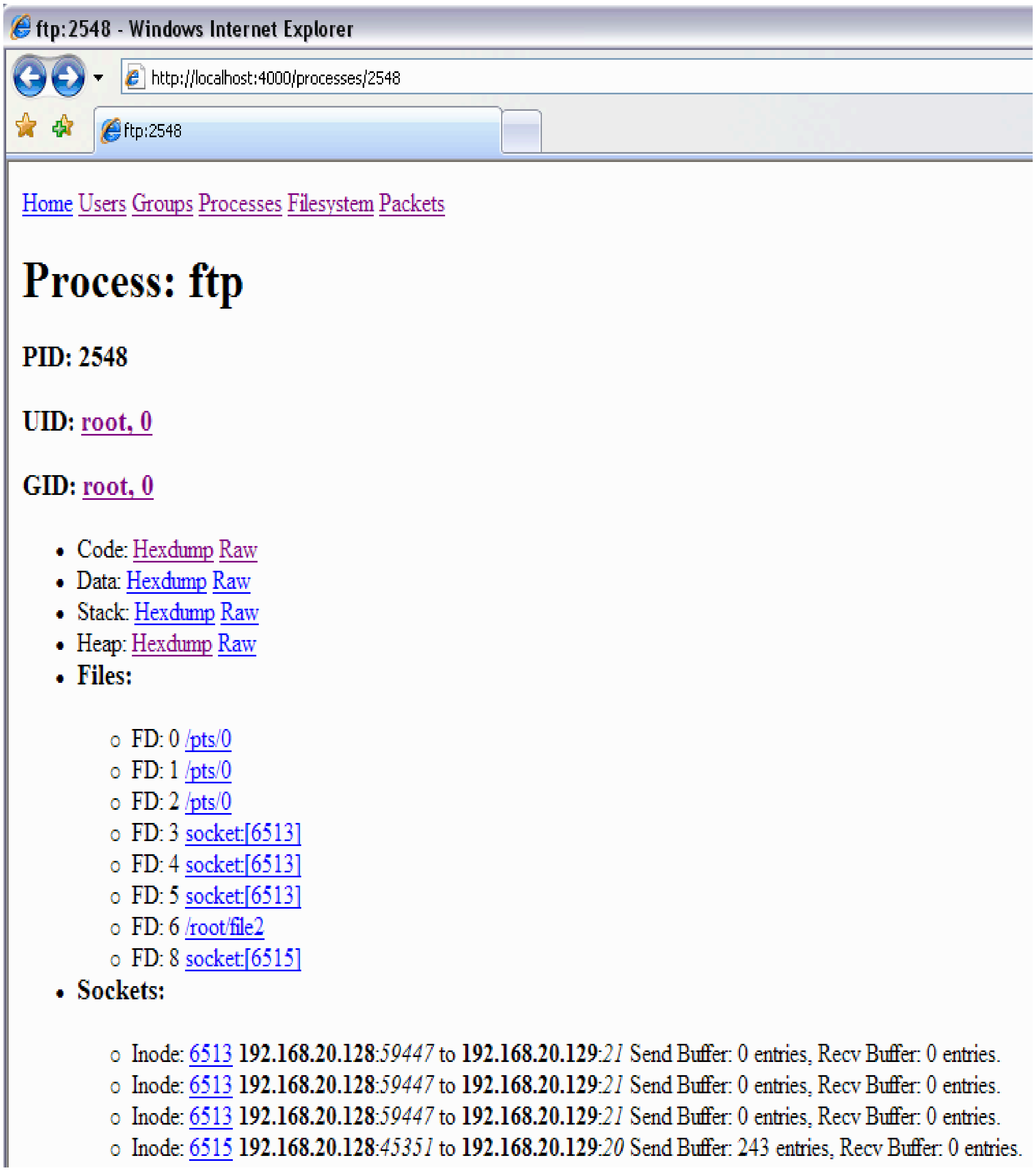
6.3.5 Sample Investigative Scenario

It is not uncommon in corporate environments for malicious users or for malware to transfer sensitive documents outside of a company’s perimeter. Using current forensics tools, the methods to determine who transferred a file and when rely on network traces or disk images that contain parts of the file or substantial fragments. This leaves significant doubt about who really transferred the file and the file’s origin. Relying solely on network traffic is ineffective because MAC addresses can easily be cloned or modified. Similarly, simply finding a sensitive file on an employee’s hard drive does not prove that the file was transferred from that machine.
We developed an experiment to test our system’s ability to correlate data from multiple sources, to provide a clear investigative picture, and to give more detailed information about the actions taken by a user. Our experiment involved transferring a large file over FTP while taking a memory dump and recording network traffic. By combining memory, disk, and network information, we hoped to map the data in the network stream back to the user who created the process on the originating machine.

Our scenario involves Ryan Acer, a network administrator for a small company who is suspected of selling trade secrets to a competitor. An outside investigative agency was secretly hired to determine if any trade secrets were being transferred outside of the company network, and if so, by whom. The investigators recorded all network traffic for a day and when they noticed an unusual spike in outbound traffic, they initiated a memory dump of Ryan Acer’s workstation. Later that day, they retrieved a copy of the Ryan’s Linux partition using dd, and used these three sources of data as input to FACE.

Two Debian version 40r3-i386 VMWare images were used, one as a user workstation, and the other as a server. The services of interest to us were the Apache 2.2.3 HTTP server and the vsFTPd 2.0.5 FTP daemon. We transferred a large file over FTP from the client machine to the FTP server while simultaneously downloading content from the web server. This mimics the activity of a user who is casually browsing the Internet while uploading sensitive documents to an outside location. The sensitive document for the experiment is named ’file2’ and contains content that is easily recognizable as proprietary. For the experiment, after the first HTTP download is complete, the client VMWare image was suspended. Performing the experiment in this manner allowed us to have a large amount of data in both the network trace and in the FTP processes' network packet queues. This also suspends the process while the FTP file connection was still open, which allows us to look at the list of open files and match data on disk with pieces of data in the socket queues and in the network stream. Running Xorg and other non-essential graphical applications while downloading the file made the experiment more realistic and produced many more processes for our tools to analyze. Wireshark recorded all traffic up to the time of the memory dump and its output was saved to a pcap file. The client file system was copied using dd and netcat. Together this data represents the complete state of the machine at the time and its network traffic.

After the scenario was enacted, we ran *ramparser* and our pre-processing scripts on the Debian memory image, the pcap network trace and the dd image of the filesystem and then FACE was started. The first step of the investigation is to discover what happened. In this scenario, the owners of the corporation believe the suspect was transferring proprietary files to an outside entity, so the first step is to look at what the user was doing. To do this, the investigator retrieves the process list using FACE. Eight processes have open network connections. Six of these are system processes and daemons, which leaves two user land processes as initial targets of investigation. The first process to be investigated is the Firefox web browser. Clicking on Firefox's entry shows that there is only one open network connection, so the investigator would look at that more closely. Upon viewing that connection's details, the investigator sees that there is relatively little network traffic, so he can quickly view the streams to see that no proprietary information was being uploaded through Firefox. Further, none of the open files appears to be proprietary data.



ftp:2548 - Windows Internet Explorer

http://localhost:4000/processes/2548

ftp:2548

Home Users Groups Processes Filesystem Packets

Process: ftp

PID: 2548

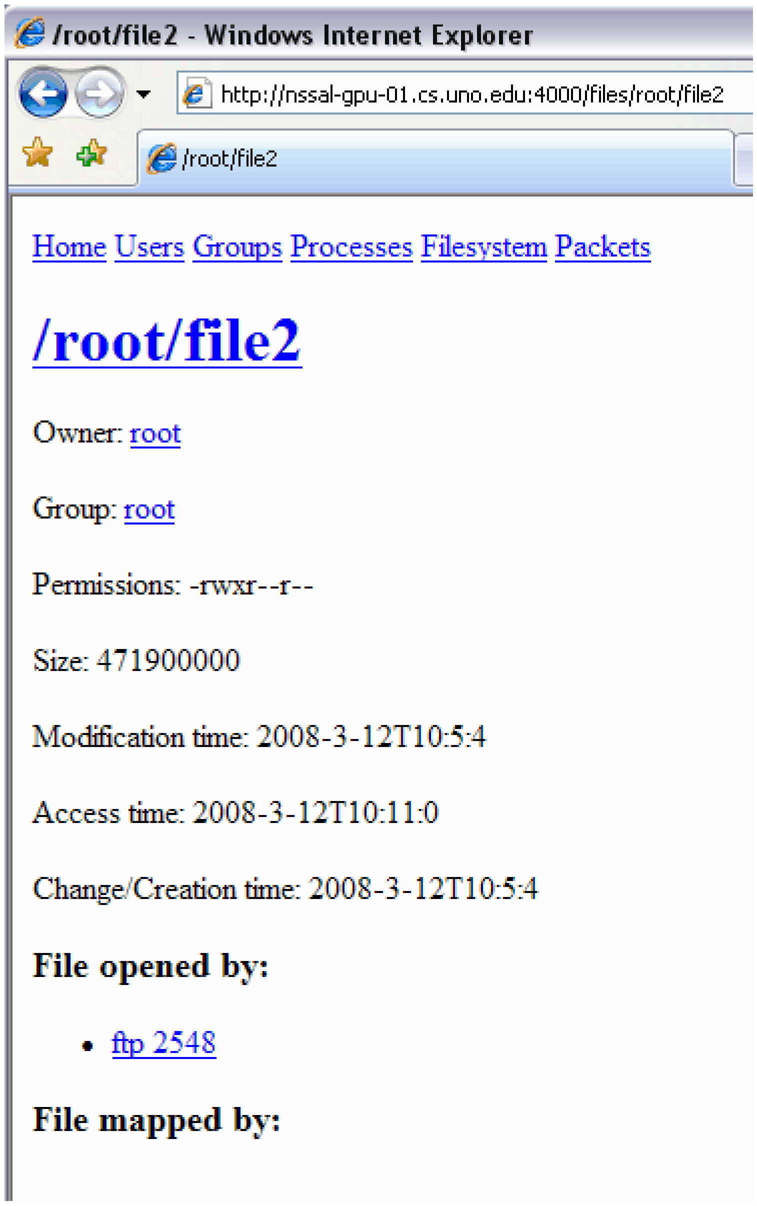
UID: root, 0

GID: root, 0

- Code: Hexdump Raw
- Data: Hexdump Raw
- Stack: Hexdump Raw
- Heap: Hexdump Raw
- **Files:**
  - FD: 0 /pts/0
  - FD: 1 /pts/0
  - FD: 2 /pts/0
  - FD: 3 socket:[6513]
  - FD: 4 socket:[6513]
  - FD: 5 socket:[6513]
  - FD: 6 /root/file2
  - FD: 8 socket:[6515]
- **Sockets:**
  - Inode: 6513 **192.168.20.128**:*59447* to **192.168.20.129**:*21* Send Buffer: 0 entries, Recv Buffer: 0 entries.
  - Inode: 6513 **192.168.20.128**:*59447* to **192.168.20.129**:*21* Send Buffer: 0 entries, Recv Buffer: 0 entries.
  - Inode: 6513 **192.168.20.128**:*59447* to **192.168.20.129**:*21* Send Buffer: 0 entries, Recv Buffer: 0 entries.
  - Inode: 6515 **192.168.20.128**:*45351* to **192.168.20.129**:*20* Send Buffer: 243 entries, Recv Buffer: 0 entries.

Figure 6.8: FTP Process View.

After noting that the user was not likely uploading proprietary data via Firefox, the investigator turns to the next likely culprit, the FTP process. Viewing this process shows two active network connections and one of them has a large amount of associated traffic and a significant amount of data left in the send queue (see Figure 6.8).


/root/file2 - Windows Internet Explorer
http://nssal-gpu-01.cs.uno.edu:4000/files/root/file2
/root/file2

Home Users Groups Processes Filesystem Packets

/root/file2

Owner: root

Group: root

Permissions: -rwxr--r--

Size: 471900000

Modification time: 2008-3-12T10:5:4

Access time: 2008-3-12T10:11:0

Change/Creation time: 2008-3-12T10:5:4

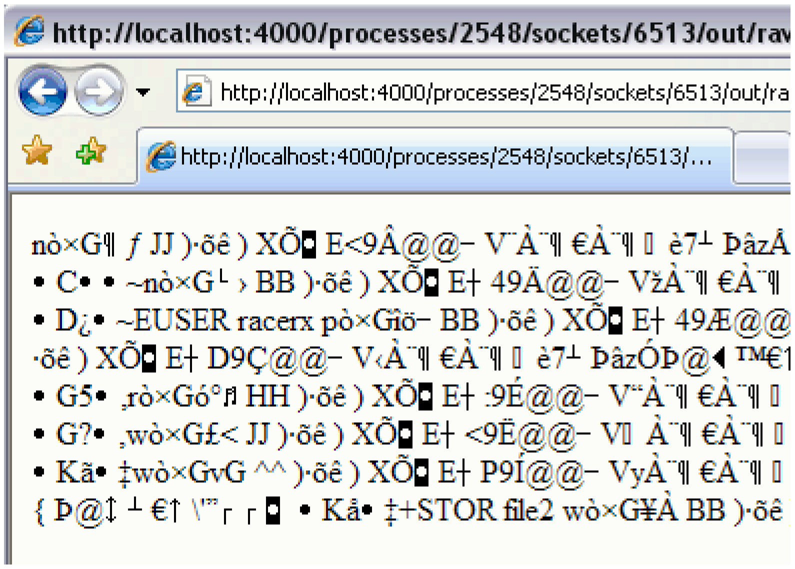
File opened by:

- ftp 2548

File mapped by:

Figure 6.9: File file2.

Further, the investigator notes that FTP has as one of its open files /root/file2 (Figure 6.9), the proprietary file mentioned in the scenario setup.



http://localhost:4000/processes/2548/sockets/6513/out/rav

http://localhost:4000/processes/2548/sockets/6513/out/ra

http://localhost:4000/processes/2548/sockets/6513/...

```
nò×G¶ ƒ JJ )·õê ) XÕ◘ E<9Â@@– V¨À¨¶ €À¨¶ ▯ è7┴ ÞâzÂ
• C• • ~nò×G└ › BB )·õê ) XÕ◘ E┼ 49Ä@@– VžÀ¨¶ €À¨¶
• D¿• ~EUSER racerx pò×Gîö– BB )·õê ) XÕ◘ E┼ 49Æ@@
·õê ) XÕ◘ E┼ D9Ç@@– V‹À¨¶ €À¨¶ ▯ è7┴ ÞâzÓÞ@◄ ™€1
• G5• ‚rò×Gó°♬ HH )·õê ) XÕ◘ E┼ :9É@@– V"À¨¶ €À¨¶ ▯
• G?• ‚wò×G£< JJ )·õê ) XÕ◘ E┼ <9Ë@@– V▯ À¨¶ €À¨¶ ▯
• Kã• ‡wò×GvG ^^ )·õê ) XÕ◘ E┼ P9Í@@– VyÀ¨¶ €À¨¶ ▯
{ Þ@↕ ┴ €↑ \"┌ ┌◘ • Kå• ‡+STOR file2 wò×G¥À BB )·õê
```

Figure 6.10: Raw Socket Data.

The investigator first views the FTP command stream, and locates the command to upload file2 (STOR file2) in the outgoing network stream (Figure 6.10). The investigator then views the contents of the data stream seeing that the proprietary information is in fact being transferred over the network as file2. Finally, the investigator clicks on the entry for open file /root/file2, and verifies the access time and the contents of the file, that it is indeed the proprietary file that the culprit should not have uploaded.

Now that the investigator knows which process was used to upload the file, he can attempt to figure out when the file was uploaded. This can be done by simply noting the times associated with the first and last packet in the FTP data connection which was used to upload the file (2008-3-12T10:10:47 and 2008-3-12T10:11:0), and can then note that this time is about the same as the last access time of the file /root/file2 (2008-3-12T10:11:0). The final question that the investigator needs to answer is whether or not the suspect was the one who uploaded the file. The FTP process was run as the root user, whose last login was 2008-3-12T9:34:57 (see Figure 6.4), about 35 minutes before the file was transferred. The login was noted to be local, so the user had to have physical access to the machine; this rules out an outside attacker. The only other clue we have to the perpetrator's actual identity is the username and password pair used in the FTP process: racerx/racerx123. Because the suspect's name is Ryan Acer, this is indicative but not conclusive evidence that he may have been the culprit.

Discussion 6.4

In this chapter we have discussed the coherence issue in DFI. Investigations of digital devices are inherently complex, and the largely ad-hoc process of synthesizing the large amounts of data generated by a similarly large set of tools is a difficult, manual process. Current tools make no attempt to integrate data generated by a set of digital forensics tools into a coherent picture of the state of the system under investigation. To remedy this situation, we developed FACE, a tool which removes some of the burden from the investigator by automatically integrating data about high-level system entities, and presenting it to the investigator as such. FACE provides an API against which modules can be constructed. These modules can take data generated by a set of digital forensics tools, and link them to the higher-level objects they describe. This linkage is presented to the investigator in a web-based interface, giving a clearer view of the state of the system. This eases the process of constructing a coherent picture of a system under investigation, which aids in conducting more effective DFI.

This chapter ends our discussion of the four characteristics of effective digital forensic investigation; the next chapter provides some conclusions and thoughts on future work.

Chapter 7:
Conclusion

Digital forensics is the science concerned with the collection, preservation and analysis of evidence from digital devices. These digital devices may include traditional computer systems, PDAs, cellular phones, and virtually any other device that stores digital data. The digital forensic investigative process proceeds in four phases. In the first phase, collection, forensically sound copies of available evidence are created. Once collection is complete, the examination phase begins, which involves enumeration of the objects of interest from available evidence. System state information, files, running processes and other types of data are parsed and catalogued for use in the next phase, analysis. The focus of the analysis phase is to use the data generated in the examination phase to answer questions regarding the investigation such as "what events have occurred on the system?" and "who was responsible for these events and when and how?" Answers to these questions may demand further evidence collection, examination, and analysis. When done, the last phase, reporting, is entered. Here the investigator generates a report detailing the findings of the investigation, along with methodology used to generate those findings.

In order to perform the essential tasks that make up these phases of a digital investigation effectively, an investigation must be *reliable*, *comprehensive*, *efficient*, and *coherent.* The current state-of-the-art in digital forensics practice falls short in each of these aspects and the focus of this dissertation has been to provide practical solutions to each, which, when taken as a whole, vastly improve the effectiveness of digital investigations. Below, each aspect of effectiveness is briefly summarized, along with the related contributions of this dissertation.

We say an investigation is *reliable* if the evidence generated is accurate, and free from tampering, whether malicious or accidental. The need for reliability in investigations is clear, without it we can not rely on any of the conclusions that are drawn. To support the reliability characteristic, investigators rely on cryptographic hashes of data, and on newer tools, such as Digital Evidence Containers. While these techniques are useful for detecting data tampering and corruption, and for bundling case metadata with case data, they do not offer help with the problem of keeping records of operations performed on the data, a process we term *auditing*. Auditing is important for two main reasons. In the course of an investigation, a large number of tools may be used in analyzing the data for evidence, and a record of the methodology used is required for report generation. Recording these activities is largely a manual, error-prone process. Second, many tools in use have bugs, and, as such, their correct function cannot be completely relied on. To remedy this situation, we developed FDAM, the Forensic Discovery Auditing Module. FDAM provides a filesystem which logs all accesses to files within it, enforces access restrictions (e.g., write blocking), and can periodically re-hash evidence. It logs not only which applications have accesses evidence files, but when, what specific parts of the evidence were written or read, and what user owned ran the application. The log created by FDAM specifically addresses the problems listed above by auditing all accesses to evidence files. In the future FDAM can be extended to record significantly more data about the context of data accesses, further increasing the reliability of investigations conducted with its use.

We say that an investigation is *comprehensive* if its analysis includes all potential items of interest. While resource constraints make this impossible in the perfect sense, a comprehensive investigation should analyze as many of the potentially interesting targets as possible. At a physical crime scene, investigators collect items which may have evidentiary value, and preserve them for later analysis. Conversely, in a digital forensic investigation, the investigator can make perfect copies of some aspects of the entire crime scene (e.g., data in non-volatile storage on the digital devices). Because digital artifacts are not plainly visible, tools are required for their analysis. This is problematic as there are many types of data which could be of forensic interest, and each requires a tool or tools for its analysis. While digital forensics currently has at its disposal a great many tools, this number is far outweighed by the number of types of data that exist on today's complex machines. This disparity underlies the many gaps where no tool exists for analysis of many interesting types of data. To fill one such gap, we developed *ramparser*, a tool for deep analysis of images of RAM on Linux systems. When a machine is powered down, much of the volatile information about its current state is permanently lost. For standard computers, this includes running processes, open files and network connections, loaded modules, and tremendous other amounts of information. Given an image of physical RAM, *ramparser* parses kernel structures to extract information on the live state of the system as of when the image was created. It is able to emulate the functions of several Linux system tools, such as *ps, lsmod,* and *netstat*, as well as do deeper extraction of process data, such as its code, data, stack, and heap. *ramparser* exposes types of information from digital devices which were simply inaccessible before its development. We plan for future versions to work across a wider range of Linux kernels, and to more fully parse available kernel structures in order to expose more data relevant to an investigation.

Intimately related to comprehensiveness, is the *efficiency* characteristic of effective digital forensic investigations. As stated above, perfectly comprehensive investigations are impossible when operating within the bounds of available resource constraints. A digital forensic investigation is efficient if it maximizes the use of constrained resources, which can include processing power, data storage, time, and manpower. The set of currently available tools, most being hastily developed to address some pressing need, are ill equipped to make efficient use of available IO, storage, multiple processor cores, or massively parallel co-processors. In order to address this issue we developed several enhancements to the open source digital forensics tool, Scalpel, a file carver, to demonstrate design decisions that should be applied to future forensic tools. File carving, a file recovery technique, is representative of many digital forensics tools in that it often deals with a large data input set (e.g., a disk image), a potentially larger output set, and can be either IO-bound or CPU-bound depending on the number of file types for which recovery is attempted, and the specific recovery techniques used. Over the course of implementing the enhancements to Scalpel, we produced order-of-magnitude-level improvements in processing speed and storage usage. We believe that theses techniques can be applied to many other digital forensic tools, achieving similar results. As part of this work, some processing was offloaded to an NVIDIA GTX260 GPU. This was the first use of this technique (offloading computation to massively parallel coprocessors) in the digital forensics community. We showed that there is tremendous potential for speeding up forensic processing using such devices, and in the future we plan to leverage this technique to develop other efficient tools.

The final characteristic of effective digital investigation covered in this research is *coherence*. We say that an investigation is coherent if the evidence resulting from the analysis can be used to compile an integrated view of the events under question. The current set of digital forensics tools is composed of a plethora of small, single purpose applications which produce data on some small part of a digital system. While some tool suites bundle many functions into one larger application, no attempt is made to provide linkages between the types of data generated by individual tools. Many individual tools can give information on a single entity in the system, e.g., a file, but linking the data together into a single picture is left to the investigator to perform manually. Take for example a document file. The *file* tool will tell the type of the file, the *Sleuthkit* will provide details of the file's allocation in the filesystem, *md5sum* will generate the hash of the file, and *Metadata Analyzer* will parse the format specific metadata stored in the file. But getting a coherent picture of all of the information about the file must be done manually by the investigator - a difficult task. We developed FACE, the Forensics Automated Coherence Engine to simplify this process. FACE provides a framework with an extensible pluggable API for integrating data from a set of tools into coherent picture of the entities in the system. Using this framework, modules can be developed which are used to absorb data from tools, and link the data to the entity the data describes. This removes much of the burden from the investigator, and aids in forming a more coherent picture of the system being investigated. Future generations of this tool will attempt to eliminate the need for module writing by examining the data output from tools and automatically deciding the linkages to create to the entities in the system. This would further reduce the burden on the investigator, and allow tool to be integrated into the framework more quickly. We also intend to research applications of this technique outside of the digital forensics arena. Specifically, the medical and geological fields, and potentially others, deal with a similar glut of low-level data from which a higher-level picture must be constructed, and some FACE-like construct may be useful.

In summary, in this research we have addressed the four characteristics of effective digital forensic investigation, reliability, comprehensiveness, efficiency and coherence, and have developed techniques to enhance each of them. There is still much work to be done, as each of the new techniques developed addresses a relatively small part of a much larger problem, but the work discussed in this dissertation makes substantial contributions to improving state-of-the-art in digital investigations.

Bibliography

[1] Carrier, B., http://www.digital-evidence.org/di_basics.html

[2] G.G. Richard III, V. Roussev, L. Marziale, "Forensic Discovery Auditing of Digital Evidence Containers," Journal of Digital Investigation, (4)2, 2007.

[3] A. Case, A. Cristina, L. Marziale, G. G. Richard III, V. Roussev, "FACE: Automated Digital Evidence Discovery and Correlation," Proceedings of the 8thAnnual Digital Forensics Research Workshop (DFRWS 2008), Baltimore, MD.

[4] G.G. Richard III, V. Roussev, L. Marziale, "In-place File Carving," Proceedings of the Third Annual IFIP WG 11.9 International Conference on Digital Forensics, 2007.

[5] L. Marziale, G.G. Richard III, V. Roussev, "Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools," Proceedings of the 7th Annual Digital Forensics Research Workshop (DFRWS 2007), Pittsburgh, PA.

[6] L. Marziale, S. Movva, G. G. Richard III, V. Roussev, L. Schwiebert,"Developing Massively Threaded Digital Forensics Tools Using Graphics Processing Units (GPUs) and Multicore CPUs," In Li, C.-T. (ed.), Handbook of Research on Computational Forensics, Digital Crime and Investigation: Methods and Solutions, IGI Global Publishing, 2009, ISBN 978-1-605-66836-9

[7] Tableau TD1, http://www.tableau.com

[8] Drive Lock, http://www.forensicpc.com

[9] Tribble, B. Carrier J. Grand, "Hardware-Based Memory Aquisition Procedure for Digital Investigations," Journal of Digital Investigation, Vol 1, No 1, 2004.

[10] SAFE Block XP, http://www.forensicsoft.com

[11] FTK Imager, http://www.accessdata.com

[12] dcfldd, http://dcfldd.sourceforge.net/

[13] mount, http://freshmeat.net/projects/util-linux/

[14] MD5, http://tools.ietf.org/html/rfc1321

[15] SHA, http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

[16] The Sleuthkit, http://www.sleuthkit.org/

[17] G. G. Richard III, V. Roussev, ”Scalpel: A Frugal, High-Performance File Carver,” Proceedings of the 2005 Digital Forensics Research Workshop (DFRWS 2005).

[18] National Software Reference Library (NSRL), http://www.nsrl.nist.gov/

[19] dtSearch, http://www.dtsearch.com/

[20] Lucene, http://lucene.apache.org/

[21] mactime, www.sleuthkit.org

[22] Norton Anti-Virus, http://www.symantec.com

[23] MacAfee, http://www.macafee.com

[24] Cain & Abel, http://www.oxid.it/cain.html

[25] Password Recovery Toolkit, www.accessdata.com

[26] FTK, www.accessdata.com

[27] Encase, http://www.guidancesoftware.com

[28] PyFLAG: Forensics and Log Analysis GUI, http://www.pyflag.net

[29] P. Turner, “Unification of Digital Evidence from Disparate Sources (Digital Evidence Bags), Proceedings of the 5th Annual Digital Forensics Research Workshop (DFRWS), New Orleans, LA, Aug 2005).

[30] P. Turner, “Selective and Intelligent Imaging using Digital Evidence Bags,” Proceedings of the 6th Annual Digital Forensics Research Workshop (DFRWS), Lafayette, IN, Aug 2006).

[31] P. Turner, “Applying a forensic approach to incident response, network investigation and system administration using Digital Evidence Bags,” Digital Investigation 4(1): 30-35 (2007)

[32] S. Garfinkel, “Disk Imaging with the Advanced Forensics Format, Library and Tools,” The Second Annual IFIP WG 11.9 International Conference on Digital Forensics, Orlando, FL, Jan 2006.

[33] S. Garfinkel, “Providing Cryptographic Security and Evidentiary Chain-of-Custody with the Advanced Forensic Format,” Library, and Tools The International Journal of Digital Crime and Forensics, Volume 1, Issue 1, January-March 2009.

[34] M. I. Cohen, S. Garfinkel and B. Schatz, “Extending the Advanced Forensic Format to accommodate Multiple Data Sources, Logical Evidence, Arbitrary Information and Forensic Workflow,” DFRWS 2009, Montreal, Canada

[35] afflib, www.afflib.org

[36] The Generic Forensic Zip Project, http://gfzip.nongnu.org/filespec.html

[37] Seekable GZIP, www.pyflag.net

[38] Encase File Format, http://sourceforge.net/projects/libewf/

[39] LinEn, http://www.digitalintelligence.com/software/guidancesoftware/encase/

[40] DIBS USA Rapid Action Imaging Device (RAID), http://www.dibsusa.com/index.asp

[41] ProDiscover Image File Format, http://www.techpathways.com/uploads/ProDiscoverImageFileFormatv4.pdf

[42] NIST, www.nist.gov

[43] Computer Forensics Tool Testing (CFTT) Project, http://www.cftt.nist.gov/

[44] S. Garfinkel, P. Farrell, V. Roussev and G. Dinolt, "Bringing Science to Digital Forensics with Standardized Forensic Corpora," DFRWS 2009, Montreal, Canada

[45] http://digitalcorpora.org/

[46] Forensic Corpora List, http://www.forensicswiki.org/wiki/Forensic_corpora

[47] Test Results for Digital Data Acquisition Tool: FTK Imager 2.5.3.14, http://www.ncjrs.gov/pdffiles1/nij/222982.pdf

[48] Test Results for Digital Data Acquisition Tool: EnCase 6.5, http://www.ncjrs.gov/pdffiles1/nij/228226.pdf

[49] Test Results for Digital Data Acquisition Tool: DCCIdd (Version 2.0), http://www.ncjrs.gov/pdffiles1/nij/220223.pdf

[50] FUSE: Filesystem In User Space, http://fuse.sourceforge.net

[51] WinHex, http://www.x-ways.net/winhex/

[52] GHex, http://live.gnome.org/Ghex

[53] xxd, http://www.vim.org/

[54] Hexedit, http://www.physics.ohio-state.edu/~prewett/hexedit/

[55] Binutils, http://www.gnu.org/software/binutils/

[56] The Foremost File Carver, http://foremost.sourceforge.net

[57] Rapier, http://sansforensics.wordpress.com/2009/01/20/rapier-a-different-data-carver

[58] fdisk, http://git.kernel.org/?p=utils/util-linux-ng/util-linux-ng.git

[59] grep, http://www.gnu.org/software/grep/

[60] tcpdump, http://www.tcpdump.org/

[61] windump, http://www.winpcap.org/windump/default.htm

[62] snort, http://www.snort.org/

[63] PCAP, http://www.tcpdump.org/

[64] Wireshark, http://www.wireshark.org/

[65] ngrep, http://ngrep.sourceforge.net/

[66] tcpextract, http://tcpxtract.sourceforge.net/

[67] dsniff, http://monkey.org/~dugsong/dsniff/

[68] nmap, http://nmap.org/

[69] netcat, http://netcat.sourceforge.net/

[70] dig, https://www.isc.org/software/bind

[71] whois, http://directory.fsf.org/project/whois/

[72] nslookup, https://www.isc.org/software/bind

[73] kismet, http://www.kismetwireless.net/

[74] Netstumbler, http://www.netstumbler.com/

[75] http://dfrws.org/2005/challenge

[76] Volatility, https://www.volatilesystems.com/default/volatility

[77] knttools, http://www.gmgsystemsinc.com/knttools/.

[78] memparser, http://sourceforge.net/projects/memparser.

[79] A. Schuster, "Searching for Processes and Threads in Microsoft Windows Memory Dumps," Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS), 2006.

[80] T. Vidas, "The Acquisition and Analysis of Random Access Memory," Journal of Digital Forensic Practice, Vol 1, No 4, pp. 315–323, December 2006.

[81] crash, http://people.redhat.com/anderson/

[82] Memoryze, http://www.mandiant.com/software/memoryze.htm

[83] Cache View, http://www.progsoc.uts.edu.au/~timj/cv/

[84] Pasco, http://www.foundstone.com/us/resources/proddesc/pasco.htm

[85] Web Historian, http://www.mandiant.com/software/webhistorian.htm

[86] Skype-Parser, http://redwolfcomputerforensics.com/

[87] P2P Marshall, http://p2pmarshal.atc-nycorp.com/

[88] Paraben's Chat Examiner, http://www.paraben-forensics.com/

[89] Process Monitor, http://technet.microsoft.com/en-us/sysinternals

[90] Access Data Registry Viewer,
http://www.accessdata.com/media/en_us/print/techdocs/Registry%20Viewer.pdf
[91] RegViewer, http://sourceforge.net/projects/regviewer/
[92] Stream Viewer, http://www.jsware.net/jsware/sviewer.php5
[93] rifiuti, http://www.foundstone.com/us/resources/proddesc/rifiuti.htm
[94] Metadata Analyzer, http://www.smartpctools.com/metadata/
[95] libpst, http://www.five-ten-sg.com/libpst/
[96] libdbx, http://freshmeat.net/projects/libdbx/
[97] PEView, http://www.magma.ca/~wjr/PEview.zip
[98] TNEF, http://sourceforge.net/projects/tnef/
[99] jpeginfo, http://www.kokkonen.net/tjko/
[100] pdfinfo, http://www.foolabs.com/xpdf/
[101] Encase Enterprise, http://www.encaseenterprise.com/products/ee_index.asp
[102] OnlineDFS, http://www.cyberstc.com/products_dfs.asp
[103] Sysinternals, http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx
[104] LiveCD's Incident Responder Collection Report (IRCR), http://tools.phantombyte.com/
[105] RAPIER, http://code.google.com/p/rapier/
[106] John the Ripper, http://www.openwall.com/john/
[107] Elcomsoft Password Recovery Bundle, http://www.elcomsoft.com/
[108] ZFS, http://hub.opensolaris.org/bin/view/Community+Group+zfs/WebHome
[109] BTRFS, http://oss.oracle.com/projects/btrfs/
[110] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," Proceedings of the 13th USENIX Security Symposium, August 2004.
[111] J. Solomon, E. Huebner, D. Bem, M. Szezynska, "User Data Persistence in Physical Memory," Digital Investigation, Vol 4, No 2, pp. 68–72, June 2007.
[112] A. Boileau, "Firewire, DMA, and Windows," http://www.storm.net.nz/ projects/16.
[113] dd, http://www.gnu.org/software/coreutils/
[114] J. Rutkowska, "Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)," BlackHat DC 2007 presentation.

[115] J. Kornblum, “Exploiting the Rootkit Paradox with Windows Memory Analysis,” International Journal of Digital Evidence, 5(1), Fall 2006.
[116] N. Ruff, M. Suiche, “Enter Sandman (Why You Should Never Go To Sleep),” PacSec Applied Security Conference, 2007, Tokyo, Japan.
[117] B. Schatz, “BodySnatcher: Towards Reliable Volatile Memory Acquisition by Software,” Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.
[118] M. Burdach, idetect, http://forensic.seccure.net/tools/idetect.tar.gz.
[119] J. M. Urrea, “An Analysis of Linux RAM Forensics,” Naval Post Graduate School Thesis, March 2006.
[120] FBI Regional Computer Forensics Lab, http://www.rcfl.gov/downloads/documents/RCFL_Nat_Annual08.pdf
[121] Intel Core i7, http://www.intel.com/products/processor/corei7/index.htm
[122] Tilera, http://www.tilera.com/news_&_events/press_release_091026.php
[123] CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf
[124] CUDA, http://www.nvidia.com/object/cuda_home.html
[125] http://www.wdc.com/en/products/products.asp?DriveID=576
[126] http://www.maximumpc.com/article/reviews/western_digital_caviar_green_2tb
[127] http://hothardware.com/Articles/Intel-34nm-X25M-Gen-2-SSD-Performance-Update/?page=4
[128] http://www.informationweek.com/blog/main/archives/2009/03/interview_intel.html
[129] http://blip.tv/file/2232410
[130] ATI Stream Software Development Kit (SDK), http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx
[131] BrookGPU, http://graphics.stanford.edu/projects/brookgpu/
[132] http://www.khronos.org/opencl/
[133] http://www.pkware.com/documents/casestudies/APPNOTE.TXT
[134] ntfs-3g, http://www.ntfs-3g.org/
[135] http://www.computerworld.com.au/article/15477/dod_seized_60tb_search_iraq_battle_plan_leak

[136] http://www.rcfl.gov/index.cfm?fuseAction=Public.N86

[137] S. Garfinkel , " Carving contiguous and fragmented files with fast object validation," in Proc. 2007 Digital Forensics Research Workshop (DFRWS) , Pittsburgh, PA , Aug. 2007, pp. 4S:2–12 **.**

[138] A . Pal, T. Sencar, N. Memon , " Detecting file fragmentation point using sequential hypothesis testing ," Digit. Investig. , to be published.

[139] M. Karresand, "Completing the Picture: Fragments and Back Again," Linköping : Linköpings universitet, 2008, 111 s. Linköping studies in science and technology. Thesis, ISBN: 978-91-7393-915-7

[140] M. Karresand, N. Shahmehri, "Reassembly of Fragmented JPEG Images Containing Restart Markers," European Conference on Computer Network Defense EC2ND in cooperation with ENISA (2008 : Dublin, Ireland) , s. 25 - 32 Los Alamitos, CA, USA : IEEE Computer Society, 2008.

[141] A. Pal, N. Memon , " Automated reassembly of file fragmented images using greedy algorithms ," IEEE Trans. Image Processing , vol. 15, no. 2, pp. 385 – 393 , Feb. 2006 .

[142] [4] A. Pal , K. Shanmugasundaram, N. Memon , " Reassembling image fragments," in Proc. ICASSP, Hong Kong , Apr . 2003, vol. 4, pp . IV–732-5.

[143] K. Shanmugasundaram, N. Memon , " Automatic reassembly of document fragments via data compression ," presented at the 2nd Digital Forensics Research Workshop , Syracuse , NY, July 2 002 .

[144] K. Shanmugasundaram, N. Memon, "Automatic Reassembly of Document Fragments via Context Based Statistical Models," Annual Computer Security Applications Conference, 2003.

[145] J. D. Kornblum, "The Linux Kernel and the Forensic Acquisition of Hard Disks with an Odd Number of Sectors," International Journal of Digital Evidence, Fall 2004, Volume 3, Issue 2)

[146] Network Block Device, http://sourceforge.net/projects/nbd/

[147] Django Web Framework, http://www.djangoproject.com/

Vita

Lodovico Marziale was born in New Orleans, Louisiana and received his B.S. in Finance from the University of New Orleans in 1999. After deciding to study Computer Science, he received his M.S. in that field from the University of New Orleans in 2006. In the fall of 2006, he entered the PhD program, and began a research assistantship, funded by the National Science Foundation. His research interests include digital forensics, operating system internals, computer security, and parallel programming. He has published several peer-reviewed research papers on these topics.