A FRAMEWORK FOR LIVE FORENSICS

BY

ELLICK M. CHAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair, Director of Research
Professor Carl Gunter
Professor Pierre Moulin
Professor Samuel T. King
Professor Alex Halderman, University of Michigan

Abstract

Current techniques used by forensic investigators during incident response and *search and seizure* operations generally involve *pulling the power* on suspect machines and performing traditional *dead box* post-mortem analysis on the persistent storage medium. These cyber-forensic techniques may cause significant disruption to the evidence gathering process by breaking active network connections and unmounting encrypted disks. In contrast, live forensic tools can collect evidence from a running system while preserving system state. In addition to collecting the standard set of evidence, these tools can collect evidence from live web browser sessions, VPN connections, IM and e-mail.

Although newer live forensic analysis tools can preserve active state, they may taint evidence by leaving footprints in memory. Current uses of live forensics in corporate incident response efforts are limited because the tools used to analyze the system inherently taint the state of disks and memory. As a result, the courts have been reluctant to accept evidence collected from volatile memory and law enforcement has been reluctant to use these techniques broadly.

To help address these concerns we present Forenscope, a framework that allows an investigator to examine the state of an active system without inducing the effects of taint or forensic blurriness caused by analyzing a running system. Forenscope allows an investigator to gain access to a machine through a forced reboot. The key insight that enables this technique is that the contents of memory on many machines are preserved across a warm reboot. Upon reboot, Forenscope patches the OS state in residual memory to kill screen savers, neutralize anti-forensics software and bypass other authentication mechanisms. We show how Forenscope can fit into accepted workflows to improve the evidence gathering process.

Forenscope fully preserves the state of the running system and allows running processes, open files, encrypted filesystems and open network sockets to persist during the analysis process. Forenscope has been tested on live systems to show that it does not operationally disrupt critical processes and that it can perform an analysis in less than 15 seconds while using only 125 KB of conventional memory. We show that Forenscope can detect stealth rootkits, neutralize threats and expedite the investigation process by finding evidence in memory.

The techniques developed in Forenscope belong to a broader class of volatile forensic techniques that are becoming increasingly important in the face of new privacy measures and changes in the way storage systems are built. We are starting to see the limitations of traditional forensic approaches emerge as users shift to using more networked services, privacy guard software and non-magnetic storage technologies such as NAND and phase change memory that do not exhibit the same data residue properties as traditional disks. These trends help motivate the development of more sophisticated forensic techniques.

To My family

Acknowledgments

I thank my advisor, Professor Roy H. Campbell, for his enduring optimism and encouragement during my years at graduate school. It was a wonderful experience learning and working with him. Credit is also due to him for reading and helping revise several other publications in addition to this thesis. The other members of my PhD committee: Carl Gunter, Pierre Moulin, Sam King and Alex Halderman provided valuable guidance and insight to foster this work.

I owe Jeffrey Carlyle and Francis David a great deal of gratitude for working with me through the years in an effective research team. We spent many late nights in the Siebel Center designing our algorithms, fixing our code, running experiments and writing papers. They are truly marvelous people to work with and are very close friends. Acknowledgements are also due to everyone in the Systems Software Research Group for their support, feedback and encouragement.

My research team was supported by many partners in industry. I am very grateful to the Siebel Scholars Foundation, Motorola, ITI, TCIP and Texas Instruments for providing us with equipment, funding and direction for our research.

I thank the many staff members in the Department of Computer Science, College of Business and ITI who ensured my smooth progress through the PhD and MBA programs. Barb Cicone, Michael Heath, Lynette Lubben, Paul Magelli, Rhonda McElroy, Anda Ohlsson, Rob Rutenbar, Bill Sanders and Tim Yardley helped me with many issues over the years relating to my research, my career and my role in the joint degree Ph.D./MBA program.

Additionally, I would also like to extend my gratitude to the vibrant Siebel Scholars community for giving me inspiration and direction. In particular, Tom Siebel has been a great leader who has always challenged the scholars to think and act on important social issues. Jenny Hildebrand and Karen Roter Davis have been supportive throughout the years both personally and for the Siebel Scholars program itself. Joshua Bennett, Shegan Campbell, Jeff Goldberg, Hanson Li, Sundar Pichai and Mark Sciortino have helped me maneuver through life by offering helpful advice.

My numerous friends in Champaign-Urbana from the MBA program and other walks of life made each day colorful and fun. These friends include Chloe Liu and Mark Ye. And finally, I will always be indebted to my family for their continuing support and love.

This research would not be possible without the invaluable contributions of Francis David, Shivaram Venkataraman, Jeff Carlyle, and Reza Farivar. The following people (in alphabetical order) contributed to beneficial discussions during the development and writing process: Rob Adams, Quentin Barnes, Sergey Bratus, Amey Chaugule, Ray Essick, Kevin Larson, Michael LeMay, Jason Lowe, Mirko Montanari, Dale Rahn, Golden Richard, Adam Slagell and Winston Wan. This research was supported by grants and fellowships from The Siebel Scholars Foundation, Motorola, ITI and DoCoMo Labs USA. We are also grateful to Motorola, DoCoMo and Texas Instruments for providing us with funding and equipment.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DOJ	Department of Justice
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
FIFO	First In, First Out
FPU	Floating Point Unit
FTK	Forensic Tool Kit
HTTP	Hyper Text Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt Request
KB	Kilobyte
MAC	Media Access Control
MB	Megabyte
MMU	Memory Management Unit
NMI	Non Maskable Interrupt
OS	Operating System
SEL	Schweitzer Electronic Labs
SSH	Secure Shell
TCB	Trusted Computing Base

TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
VPN	Virtual Private Network

Chapter 1

Introduction

Thesis Statement:

Forenscope provides high-fidelity live forensics that extract residual evidence interactively without inducing taint.

Current forensic tools are limited by their inability to preserve the hardware and software state of a system during investigation. Post-mortem analysis tools require the investigator to shut down the machine to inspect the contents of the disk and identify artifacts of interest. This process breaks network connections and unmounts encrypted disks causing significant loss of evidence and possible disruption of critical systems. In contrast, live forensic tools can allow an investigator to inspect the state of a running machine without disruption. However existing tools can overwrite evidence present in memory or alter the contents of the disk causing forensic *taint*. Furthermore, taking a snapshot of the system can result in a phenomenon known as forensic *blurriness*[1] where an inconsistent snapshot is captured because the system is running while it is being observed. These shortcomings adversely affect the fidelity and quantity of evidence acquired and can cast doubt on the validity of the analysis, making the courts more reluctant to accept such evidence [2].

Experts at the SANS institute and the DOJ are starting to recognize the importance of volatile memory as a source of evidence to help combat cybercrime [3, 4]. In response, the SANS institute recently published a report on volatile memory analysis [5]. To help address the limitations of existing volatile memory analysis tools we present Forenscope, a framework for live forensics, that can capture, analyze and explore the state of a computer without disrupting the system or tainting important evidence. Chapter 2 shows how Forenscope can fit into accepted workflows to enhance the evidence gathering process.

Many elements of a computer's state are kept in volatile memory. Information such as running processes, open network connections, browser sessions and mounted encrypted disks are only available while a computer is left running and logged in. When a forensic analyst encounters a suspect system that is still running, he or she is often faced with the choice of whether or not to "*pull-the-plug*." By cutting power to the machine, the forensic analyst implicitly makes the choice to perform *dead box analysis*, which studies the contents of a computer's persistent storage devices such as hard drives. On the other hand, by making a conscious choice to leave the machine running, the analyst has opened the door to *live forensics*. This thesis is focused on the development of *forensics techniques* to analyze residual evidence in a computer's volatile memory.

While live forensics can potentially offer vast amounts of information about the ephemeral state of the system, such techniques are still relatively nascent and often cannot produce repeatable results due to the volatile nature of the problem and presence of unpredictable external inputs. Unlike dead box analysis, live forensics is faced with the risk of causing distortive alterations to the computer's state. Existing tools to analyze a live system are generally concerned with obtaining a snapshot of memory and system state rather than allowing the investigator to freely explore the state of a system while it is running. Such exploration can allow an analyst to collect additional evidence across open network connections, VPN sessions and browser sessions before the system is taken down. While live tools can run the risk of alteration, they are often times the only avenue by which an investigator can recover certain kinds of key information mentioned earlier. Without such information, it would be significantly harder to gather convincing evidence to lead to a successful conviction.

Forensic investigators often have two conflicting requirements alluded to in our discussion of dead box vs. live forensics. They need to be able to explore the state of a system interactively to look for ephemeral evidence, but they also need some level of assurance that the system under investigation isn't exposed to an excessive amount of distortive alterations. Forenscope's design provides the ability for investigators to make fewer compromises in the investigation process. To illustrate how the tool can help, we describe a typical scenario in Chapter 2 using Forenscope and compare it against what an investigator may do using conventional tools.

Forenscope leverages DRAM memory remanence to preserve the state of the running operating system across a ”state-preserving reboot” (Chapter 5), which recovers the existing OS without going through the full boot-up process. This process enables Forenscope to gain complete control over the system and perform taint-free forensic analysis using well-grounded introspection techniques from the virtual machine and simulation community[6]. Finally, Forenscope resumes the existing OS, preserving active network connections and disk encryption sessions while causing minimal service interruption in the process. Forenscope captures the contents of system memory to a removable USB device and activates a software write blocker to inhibit modifications to the disk. To maintain fidelity, it operates exclusively in 125 KB of unused legacy conventional memory and does not taint the contents of extended memory. Since Forenscope preserves the state of a running machine, it is suitable for use in certain production and critical infrastructure environments. To show its versatility, we have thoroughly tested and evaluated Forenscope on an SEL-1102, a power substation industrial computer, and an IBM desktop workstation. The machines were able to perform their duties under a variety of test conditions with minimal interruption and running Forenscope did not cause any network applications to time out or fail. To enable customized workflows, Forenscope allows forensic analysts to develop domain-specific analysis modules. We have implemented several modules that can check for the presence of malware, detect open network sockets and locate evidence in memory such as rootkit modifications to help the investigator identify suspicious activity.

The contributions of this work include:

1. An extensible software framework for high-fidelity live forensics that conforms to the best practices of a legal framework of evidence.
2. Efficient techniques to gather, snapshot and explore a system without bringing it down.
3. An analysis of the characteristics of evidence preserved in volatile memory.
4. Implementation and evaluation on several machines including a standard industrial machine and against several anti forensics rootkits.

The rest of this thesis is organized as follows: first, we introduce cyber-forensics in Chapter 2 and characterize evidence in residual memory in Chapter 3. Chapter 4 describes Forenscope, Chapter 5 describes the design, Chapter 6 discusses forensic modules and we evaluate the effectiveness of Forenscope in Chapter 7. Chapter 8 discusses issues involved in forensics, Chapter 9 surveys related work and we conclude in Chapter 10.

Chapter 2

Background

2.1 The Forensic Process

2.1.1 Definitions of Forensics

Forensics is a broad field that applies the use of scientific methods and techniques to the investigation of crime. Although the term “digital forensics” can have many definitions depending on the context, we adopt a commonly accepted definition of digital forensics as: *Tools and techniques to recover, preserve, and examine digital evidence on or transmitted by digital devices*

Other definitions currently in use include:

1. *“Information stored or transmitted in binary form that may be relied upon in court” [7]*
2. *“Information of probative value that is stored or transmitted in binary form” [8]*
3. *“Information and data of investigative value that is stored on or transmitted by a computer” [9]*
4. *“Any data stored or transmitted using a computer that support or refute a theory of how an offense occurred or that address critical elements of the offense such as intent or alibi” [10]*

2.1.2 Locard’s exchange principle

Locard’s exchange principle states that “with contact between two items, there will be an exchange.” This principle is more commonly known as the idiom “every contact leaves a trace”[11]. It is one of the foundational principles behind forensics that suggests that when a crime is committed, evidence will be left behind no matter how much the culprit tries to clean up his tracks. This link is fairly evident in the physical world. For instance, a bullet fired from a gun will leave behind an imprint of the gun’s barrel on the surface of the bullet as it leaves the barrel. This imprint can be used to trace back to the gun that fired the bullet. Likewise, in the digital world, log files leave records of a suspect’s actions. Even if a suspect successfully erases or alters a system’s logs, other forms of evidence such as file modification time or the use of multiple log files may still suggest a suspect’s culpability. In the context of this thesis, the contents of volatile memory [12] have been used as evidence where traditional forms of evidence such as log files did not provide sufficient information.

Locard’s principle of exchange works both ways: it can incriminate a suspect with evidence left behind, but if an investigator is not careful in collecting the evidence, he may cause an undesired exchange and accidentally contaminate the original crime scene’s evidence by inducing his own changes. Therefore, it is critical for investigators to follow procedures that preserve evidence and minimize taint.

2.1.3 Digital Forensic Process

The U.S. National Institute of Justice (NIJ) has published a process model in the Electronic Crime Scene Investigation Guide [13]. Their workflow is meant to help guide a first responder and it consists of the following steps:

1. Preparation: Prepare the equipment and tools to perform the tasks required during an investigation.
2. Collection: Search for, document, and collect or make copies of the physical objects that contain electronic evidence.
3. Examination: Make the electronic evidence visible and document contents of the system. Data reduction is performed to identify the evidence.

4. Analysis: Analyze the evidence from the Examination phase to determine the significance and probative value.
5. Reporting: Create examination notes after each case.

In this work, we are primarily concerned with the collection phase. The fidelity of evidence throughout the workflow must be preserved to show that the alleged evidence is in fact related to the alleged crime. This process is called the **Chain of custody**. To illustrate this concept, suppose that an investigator collects a blood sample at a crime scene. This blood sample is then sent to a lab for analysis. Upon completion of testing, the sample is sent to a warehouse for archival purposes. Then, suppose the defendant challenges the authenticity of the evidence and requests proof of validity. The investigator must show that the blood sample analyzed was indeed the sample found at the crime scene. The evidence handling process requires the investigator to document each action carefully to show that the sample collected at the crime scene is authentic and that each transfer of the sample to another party did not result in contamination or misidentification. Chain of evidence in the physical sense often involves specific procedures and documentation. In the cyber sense, it often involves taking a cryptographic hash of the evidence collected. If the hash values do not match, the evidence cannot be admissible in a court.

Many states use the Daubert guidelines [14] as a measure to prevent the use of “junk science” in court. These guidelines specify that a forensic procedure must adhere to the following guidelines to be admissible in court:

1. Published - The procedure in question must be published
2. Community acceptance - The community must accept the use of this procedure
3. Tested - The procedure must be tested to ensure that it produces the desired results
4. Error rate - The error rate of the procedure must be known

With regard to Forenscope, our methodology is published in the academic literature and we believe that it will gain the community's acceptance in time. We attempt to do our best to ensure that our procedure is tested and we measure our error rate in Chapter 7.

2.1.4 Is forensics a science?

Traditionally, forensic science has been concerned with forensics in the physical world. In this context, investigators have the benefit of working with known and predictable concepts, which allow them to predict the effects of physical laws such as that of gravity without having to measure or rediscover it each time at the crime scene. In the digital world, such laws rarely exist, and if they do, they are often highly dependent on the operating environment.

Loosely speaking, the term "forensic science" has been frequently used [15], but it has not been clear whether these techniques are actual applications of scientific or of engineering principles. Much of the known academic work in "digital forensics" thus far has been based upon ad hoc techniques to recover data from storage devices and volatile memory. To the best of our knowledge, the most significant attempt to formalize these processes is Brian Carrier's doctoral dissertation [16]. We use Carrier's formalism throughout this dissertation to help formally reason about our own work. Carrier has successfully used this formal description to categorize and reason about many well-known forensic techniques.

2.2 Formal Definition for Forensics

Carrier[16] uses a finite state machine (FSM) to model a digital crime scene that may consist of a computer, storage devices and the attached network. As this system operates, it reacts to a stream of incoming events that advance the state of the machine. In his model, the state of the machine is defined as the state of all storage devices. Peripheral devices such as hard drives are simply storage devices and networks are considered a special case where the network cable is a temporary storage device and the computers at the other end are also finite state machines. By representing the lowest levels of the machine as a state machine, Carrier is able to compose these abstractions into layers to define data structures, the execution of software and the forensic process itself. We use his formalism in this dissertation to represent our concepts more precisely.

We briefly summarize Carrier's FSM as follows:

The state machine is defined by the quintuple $M = (Q, \Sigma, \delta, s0, F)$ where Q is a finite set of machine states and Σ is a finite alphabet of event symbols. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is the event mapping between states in Q for each event symbol in Σ. The machine state changes only as a result of a new input symbol. The starting state of the machine is $s0 \in Q$ and the final states are $F \subseteq Q$ [17].

To complement the state machine model, Carrier also defines a model for observations. Since the investigator cannot physically observe the state of the machine, he must rely on the operating system and software processes to query the state of the machine. If any part of this system is not trustworthy, for instance, if it has been compromised by malware, then the output cannot be trustworthy. Carrier's model uses an information flow graph $G = (V, E)$ where V is a set of vertices and E is a set of directed edges. Each component is a vertex and an edge exists from vertex a to vertex b if information flows from a to b. Figure 2.1 depicts this relationship. In this figure, information flows from the bottom-most layers up to the investigator. Events observed by the investigator must have a valid chain of trust to be trustworthy. For instance, if the flow from the disk to the editor was tapped by a rootkit, an attacker may be able to conceal critical files or processes from an investigator.

2.2.1 Deadbox vs. Live Forensics

Traditionally, dead box forensics referred to physically shutting a machine down and performing a post-mortem analysis on the persistent storage medium. The "dead" designation refers to the fact that the analysis is performed while the target OS is not being executed

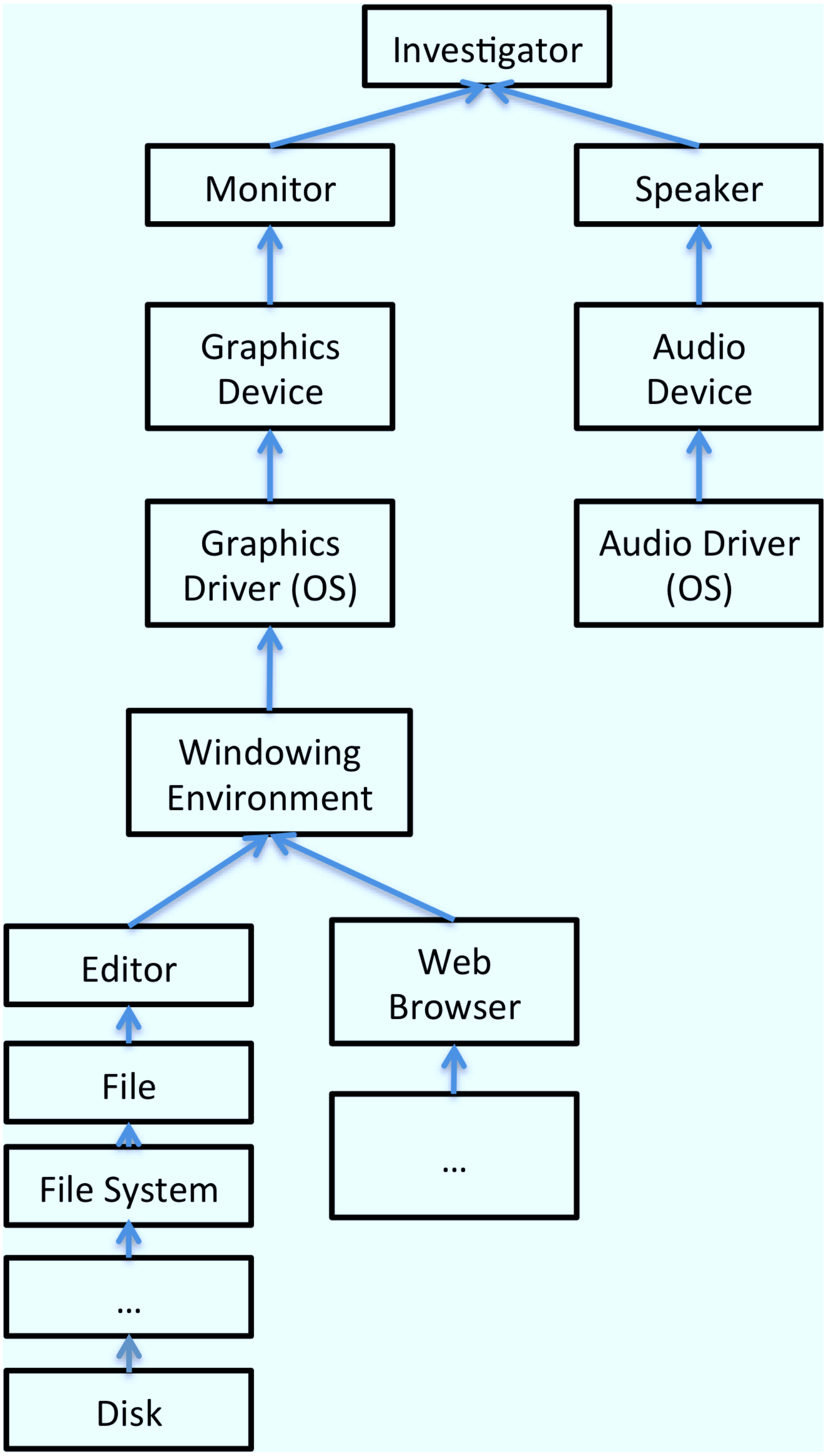

Figure 2.1: Information flow for components

or used to analyze the data. Instead, a separate boot medium such as a trusted boot CD is used.

In contrast, live forensics techniques do rely on the target operating system to provide services to perform the analysis. Although it's convenient to distinguish these techniques based on whether the target machine is powered on or off, we think that an alternate perspective based on trustworthiness is helpful for this dissertation. In this perspective, dead box forensics inherently distrusts the OS and software stack, so the forensic tool provides its own analysis environment. Live forensics in this case then relies on existing software facilities to perform the analysis. Recall the trusted information flow paths in Figure 2.1. Dead box forensics formally does not depend on any nodes in the graph that may belong to the incumbent OS. Live forensics must share one or more nodes in the information flow graph to acquire information.

The target of this dissertation is to develop a hybrid tool, Forenscope, which has the trustworthiness benefits of dead box analysis and the uptime benefits of live analysis. Forenscope achieves this desired state by providing a recoverable "golden state" that exhibits the trustworthiness properties of dead box analysis, and this state can be recovered to the original running system afterward to provide the benefits of live analysis. Chapter 4 describes this state in more detail.

2.3 Existing Digital Forensic Tools

Digital Forensic investigators often have two conflicting requirements alluded to in our discussion of dead box vs. live forensics. They need to be able to explore the state of a system interactively to look for ephemeral evidence, but they also need some level of assurance that the system under investigation isn't exposed to an excessive amount of distortive alterations. Forenscope's design provides investigators with several features that allow an investigator to make fewer compromises in the investigation process.

While the steps described in Section 2.1.3 are executed for most cases, there are special requirements for each case. For instance, in criminal investigation, the integrity and fidelity of the data is paramount. As evidence presented in court must be as accurate as possible, special steps must be taken to ensure fidelity. For incident response, the goal is to detect and react to security breaches while minimizing the intrusiveness of the process. In some critical systems it is impractical to interrupt the system to perform forensic analysis of a potential breach and service level agreements (SLAs) may impose financial penalties for downtime. To preserve the fidelity of the original evidence, many forensic workflows capture a pristine image of the evidence and draw conclusions based on analysis of the copy. Conventional post-mortem forensic workflows perform this task by physically shutting down a computer and copying the contents of the hard drive for subsequent analysis. On the other hand, live forensics are often desired during evidence collection because they provide access to networked resources such as active SSH and VPN sessions, remote desktop connections, IM clients and file transfers. However even state-of-the-art solutions often cannot image a system with high fidelity and frequently introduce taint in the process. In summary, existing tools require the investigator to make a trade-off between an increase in fidelity through post mortem analysis and the potential to collect important volatile information using live forensic tools at the cost of tainting evidence.

One of the key issues in collecting volatile information is that various forms of data such as CPU registers, memory, disk and network connections have different lifetimes. To maximize evidence preservation, RFC 3227 [18] outlines the *order of volatility* of these resources and dictates the order in which evidence should be collected for investigation. Commercially available products currently used by forensic experts for incident response such as Encase [19], Microsoft COFEE [20], Helix [21], FTK Imager [22], Volatility [23] and Memoryze [24] etc., do not capture all forms of data. A comparison of these products is presented in Table 2.1. Scalpel[25] and Sleuth kit are solely designed for disk analysis while other tools such as Encase, Helix and FTK include some level of memory capture

Table 2.1: Comparison of Forenscope with existing forensic tools

Evidence	Registers	Memory	Network	Processes	Disk	Encryption
RFC 3227 Reqs	Nanosecs	Seconds	Minutes	Minutes	Hours	Hours
Encase	×	✓	×	×	✓	×
Helix	×	✓	✓	✓	✓	×
FTK	×	✓	✓	✓	✓	✓
Scalpel	×	×	×	×	✓	×
Memoryze	×	✓	✓	✓	×	×
Volatility	×	✓	✓	✓	×	×
MS COFEE	×	✓	✓	✓	×	×
Volatility	×	✓	✓	✓	×	×
Sleuth kit	×	×	×	×	✓	×
Forenscope	✓	✓	✓	✓	✓	✓

and analysis capability. Memoryze and Volatility are the only tools listed in the table that attempt to perform volatile memory analysis. Some tools such as Helix, FTK and Memoryze can list the state of open network sockets, but the underlying network connections may not be preserved during the analysis process. All live forensic tools listed in this table rely on the integrity of the running kernel. Compromised systems may provide inaccurate information. Evidence preservation and minimizing forensic intrusiveness are hard problems that haven't been adequately addressed in the literature.

In contrast, Forenscope was built to comply with the collection stage where it maximizes the preservation of evidence and avoids disruption of ongoing activities to allow the capture of high fidelity evidence. As a result, we believe that Forenscope may be more broadly applicable to various scenarios that require live forensics such as incident response and criminal investigation. For incident response, we recognize that the integrity of the machine may be violated by malware and our solutions have been designed to address this scenario. For criminal investigation, we presume that the machine may have various security mechanisms implemented such as encrypted disks coupled with authentication mechanisms such as logon screens and screensaver locks.

Quantity	Description
Snapshot $\overline{S_t}$	Instantaneous contents of memory at time t
Natural drift δ_v	The change in the state of system over time t which can also be referred to as the blurriness of the system
Captured snapshot $\hat{S}_v$	Contents of captured memory snapshot with v being the time taken to capture the snapshot
Taint f	f is defined as the memory taint caused by the forensic introspection agent

Table 2.2: Definitions

2.4 Taint and Blurriness

Although existing tools and techniques have trouble with capturing certain forms of volatile information, they are also subject to capturing inconsistent snapshots taken while the system is running. To quantify these effects, we measure forensic taint and blurriness.

Taint and blurriness are concepts related to the use of forensic tools. Taint is a measurement of change in the system, in-memory or on-disk, induced by the use of a forensic tool. In this section, we only consider in-memory taint because BitBlocker (Chapter 6) eliminates disk taint by blocking writes. Blurriness refers to the inconsistency of a memory snapshot taken while a system is running.

These quantities can be reasoned about using the formalism described earlier. As a system's state S_i changes in response to operating system event E_{OS} to S_{i+1}, a forensic imager capturing the contents of memory would take an inconsistent snapshot over the capture interval due to the arrival of these events. The captured image would be a blend of the states visited during the capture process. Likewise, running a forensic acquisition tool may induce an event E_F to transition the state from $S_i \to S_{i+1}$. Since this transition was induced by an external event from the forensic investigator to capture the contents of memory deliberately, we consider this state change as one that induces taint.

More precisely, let $\overline{S_t}$ be the contents of memory at any given instant of time t. The state of a system changes over a period of time due to the natural course of running processes and we define this as the natural drift of the system, δ. When a traditional live forensic tool attempts to take a snapshot of the system, there is a difference between what is captured, $\hat{S}_v$ and the true snapshot $\overline{S_t}$, where v represents the time taken to capture the snapshot. There are two reasons for this difference: the first being δ_v the natural drift over the time period when the snapshot was being acquired (v) and the second due to the footprint f of the forensic tool. We define the former as the blurriness of the snapshot and the latter quantity to be the taint caused by the forensic tool. Table 2.2 captures these definitions in a concise form. In general, there are two ways to obtain a snapshot of the machine's state: active techniques and passive techniques. Active techniques involve the use of an internal agent on the machine, such as Encase or *dd*, which may leave an undesired footprint. Passive techniques operate outside the domain of the machine and do not affect its operation, one such example is VM introspection. When a passive acquisition tool is used, the relationship $\hat{S}_v = \overline{S_t} + \delta_v$ indicates that the approximate snapshot differs from the true snapshot due to the blurriness δ_v. In contrast, when an active forensic tool is used, $\hat{S}_v = \overline{S_t} + f + \delta_v$, where f represents taint and δ_v represents blurriness. Collectively, these quantities are a measure of the error in the snapshot acquisition process. Taint can result from the direct action of forensic tools or indirect effects induced in the system through the use of these tools. We call the former first-order taint and the latter second-order taint. First-order taint can result from loading the forensic tool into memory and second-order taint can result from system effects such as file buffering due to the effects of a forensic tool writing a file to disk.

Chapter 3

Measuring the Volatility of Evidence

To quantify more precisely the residual evidence available and the effects of blurriness, we developed a tool to instrument how applications use memory. Our tool, Cafegrind[1][26], is an extension to the Valgrind memory checker. Cafegrind instruments all memory accesses and allocation activities. Using this tool, we were able to determine that a significant corpus of applications exhibit data retention properties desirable for volatile memory forensics.

We ran Cafegrind on several key applications including Firefox, Tor and Konqueror. Our results suggest that these applications can hold potential evidence for quite a lengthy period of time, even when privacy modes such as private browsing are activated. As a result, there is a wealth of evidence to be gathered by using these methods. We briefly summarize the key findings of Cafegrind here and a more detailed discussion of Cafegrind can be found in Appendix A. Chapter 7 presents a more thorough evaluation of the taint and blurriness characteristics of our tools against the state of the art.

Cafegrind quantifies empirically the longevity and blurriness of evidence by instrumenting creation/deletion and access/modification events. As a program runs, Cafegrind determines the type of allocated objects on the fly by using the type inference algorithm described in Appendix A. This information is used to build a live object map of all active objects in memory and their links to other objects. Each allocated object is then carefully watched by the framework to monitor for accesses and modifications as the program is running. Due to efficiency reasons, we only track object-level accesses instead of member-level accesses, although the latter can be done with a small amount of effort albeit with a

[1] Cafegrind: **C**/C++ **A**nalysis **F**orensic **E**ngine for (Val)grind

Quantity	Description
Internal blurriness	Internal uncertainty of an object due to object updates. This is measured in terms of write velocity and write fraction.
Program lifetime (Ω)	Time interval from program start to termination
Natural object lifetime (ω)	Time between malloc() and free() for an allocated object
Residual object lifetime/Freed age(τ)	Time between free() and the first write that clobbers any byte of a freed object
Write velocity (λ)	The rate of change of an object defined as $\frac{\#writes}{\omega}$
Write fraction (α)	A measure of the relative frequency of writes. Defined as $\frac{\#writes}{\#reads+\#writes}$
Key diagonal	Diagonal in freed age vs. age plots where $\omega + \tau = \Omega$; these objects persist for the lifetime of the program.

Table 3.1: Internal blurriness definitions

performance penalty.

These measurements enable us to quantify object lifetime and modification rate. We define natural object lifetime as the span of time between an allocation and deallocation request. In contrast, residual object lifetime is the amount of time between a deallocation request and subsequent destruction of residual object data caused by writing new data to the recycled memory location.

Write Velocity, λ, is a measure of internal forensic blurriness(as opposed to systemic external blurriness). Internal blurriness, in contrast to external blurriness defined in Section 2.4, is a measure of the rate of change of an object. We assume that most objects are small enough to be captured instantly, so inconsistency due to delays in capturing are relatively small. This quantity provides a measure of how frequently data is updated in an object and how much historic data may have been destroyed by the process of updating. For instance, this is important when assessing the fidelity of data found in a cache object. Some entries in the cache will be very old while others may be fresh. The "internal blurriness" of the cache object provides a measure of these qualities. Likewise, blurriness can be defined for types where an aggregate summary of blurriness is measured for all instances of a type. To quantify internal blurriness further, we also measure the fraction of write accesses to data structures called the **Write Fraction**, α. This quantity measures the percentage of write accesses to a data structure and it is defined as the number of writes divided by the total number of accesses (reads+writes). A write fraction of 0 means that the structure is never initialized or written to. In contrast, a write fraction of 1 implies that the structure is never read. For some messaging buffers, a write fraction of 0.5 can be expected as the message is written once and read once. Smaller numbers imply that a structure is more read-heavy, while larger numbers imply that an object is updated frequently. Objects with a high write fraction exhibit larger amounts of internal blurriness. Table 3.1 summarizes these quantities.

Empirically, we have found that locks tend to be slightly read-heavy since a lock is only written to when it is successfully acquired, however it can be read many times as clients attempt to acquire the lock. We also found that some animated UI elements can be updated more frequently than the global UI renders the state and incoming input/network buffers can drop events when they are overloaded, so a written data item may never be read from the ring buffer under these circumstances making these structures generally write-heavy. On the other hand, read-heavy structures include program code and global settings such as the time zone that are written once and frequently accessed.

3.1 Applications

Using these volatility metrics, we studied the following applications and their data types to characterize the emergent behavior of residual evidence.

Firefox

Firefox is a popular web browser that supports a private browsing mode. When Firefox starts up, it allocates a number of singleton UI and bookkeeping structures. As the user opens web pages, Firefox creates a number of HTML parsers using the Gecko rendering engine [27], XML parsers, UI widgets and graphical image renderers for PNG/JPG images present on a webpage. The purpose of our study is to characterize the lifetime of these ephemeral elements. This measurement provides a rough estimate on the types and quantities of available forensic evidence.

Figure 3.1 shows a histogram of the distribution of object ages for a single run of Firefox. Our test loads http://www.wsj.com and we close the browser as soon as the page finishes loading. Many of the objects allocated by Firefox have a long lifespan. This is likely to be the case for several reasons. First, many of the elements created by Firefox are UI elements that last for the duration of the browsing session. Secondly, Firefox is very conservative in cleaning up old objects because the user may elect to navigate backward in the page history. This behavior has also been reported in [28]. Thirdly, Firefox uses a custom allocator and smart pointers to manage reference counts of shared objects. Therefore the lifetime of an object cannot be easily determined. This is because page elements are shared with the JavaScript engine and web plugins and elements cannot be destroyed

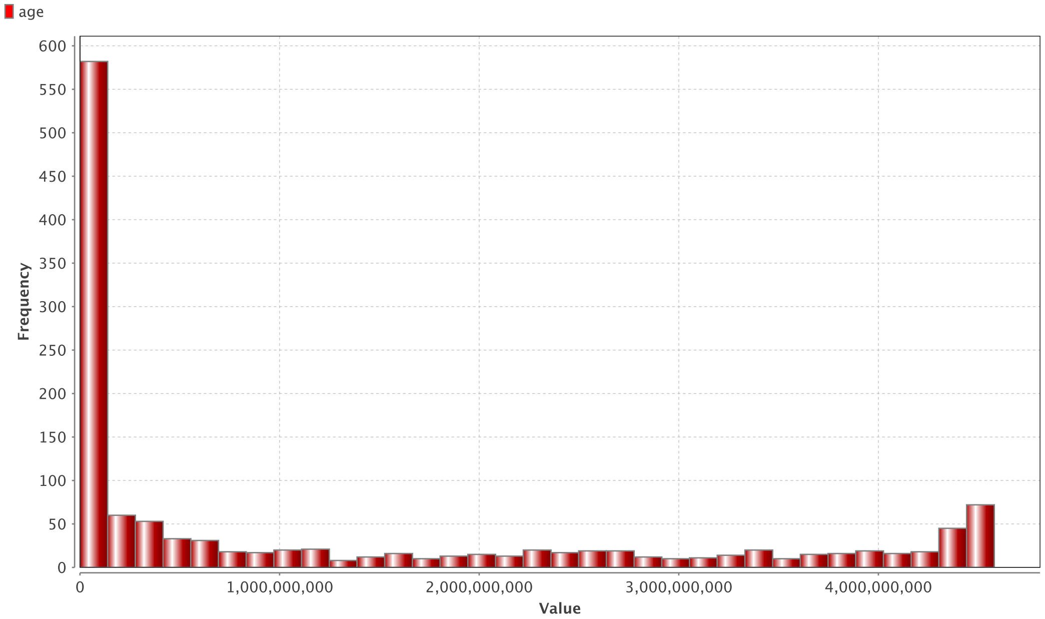

Figure 3.1: Firefox: Object age histogram (Valgrind ticks)

while they are in use. To meet these design requirements, objects are managed by using a reference counting approach. Finally, the histogram appears relatively flat because the instrumented browser renders objects very slowly. Cafegrind instrumentation added extra overhead to Firefox to slow the page load time from 3-5 seconds to around 6-10 minutes, so the histogram has been dilated horizontally.

Outside of private browsing mode, we found the structures in Table 3.2 to be key to the operation of Firefox. The list is extensive, so we present an abbreviated list of the most important types. The items in this table were selected because to highlight specific data strucutes that are likely to contain evidence. These structures have one or more of the following properties: longevity, high ASCII content, or high entropy.

Tor

Tor is a privacy guard tool that anonymizes web sessions by using an “Onion routing” network that hides the identity and location of the user behind a sophisticated peer to peer network. When a client makes a network request, Tor encrypts the request in a layered message so that intermediate routers cannot access the contents of the message. Only the originating host and the “exit node” can access the plaintext data. The exit node is the final node in the network route where the network flow connects to the wider Internet. Packets flowing through this node are not protected by Tor and thus this presents an opportunity for forensic software to monitor a connection’s plaintext content.

From a forensic perspective, we are interested primarily in latent information that may have been persisted as part of a past request. This scenario is useful because a suspect may boot off a non-persistent bootable CD to specifically run Tor without persisting any evidence to disk. In this case, the only evidence available may be found in a memory dump.

Our analysis confirms that Tor can persist information while it is operating and this latent information may contain historic evidence revealing the browsing habits of the user and any traffic that passes through the exit node.

Konqueror

Konqueror is a Linux file manager and web browser based on the popular WebKit framework [29] used by many browsers such as Google Chrome, Apple Safari and Mobile Safari. We tested Konqueror because it uses Webkit under the hood and any insights gained from this analysis are likely to be applicable to other Webkit-based browsers.

3.1.1 Memory allocator behavior

Doug Lea's malloc() allocator in GLIBC 2.x [30] is a balanced memory allocator implementation that fares well in terms of speed, space efficiency and latency. This allocator tends to have the following properties(documented in the source code): small allocations are made from a pool of quickly recycled chunks, large allocations ($\geq$ 512 bytes) are made in FIFO order, and very large ($\geq$ 128 KB) are made directly using system memory facilities such as *mmap*. These policies are encouraging for data recovery since large buffers and cache objects tend to be allocated in large pools that aren't frequently recycled.

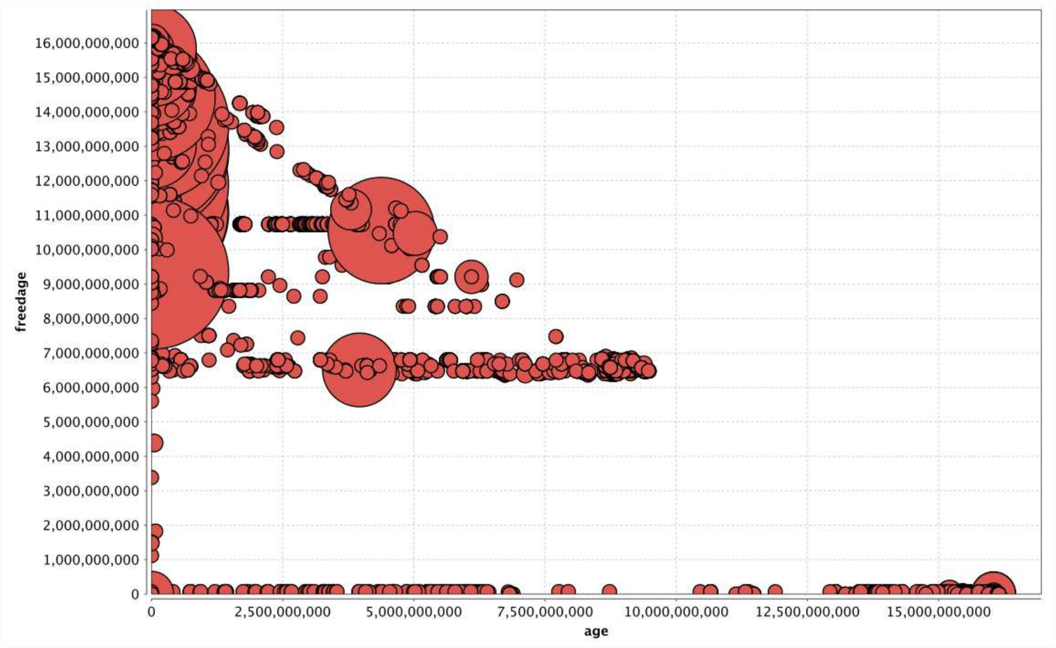

Figure 3.2: Firefox: Age vs. Freed age bubble figure

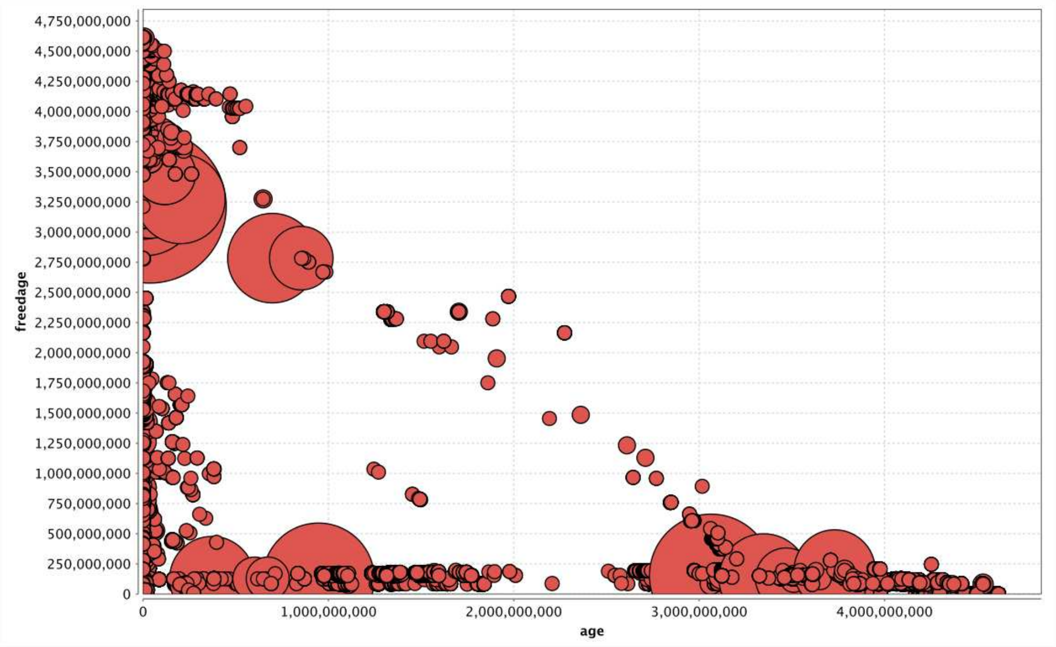

Figure 3.3: Konqueror: Age vs. Freed age bubble figure

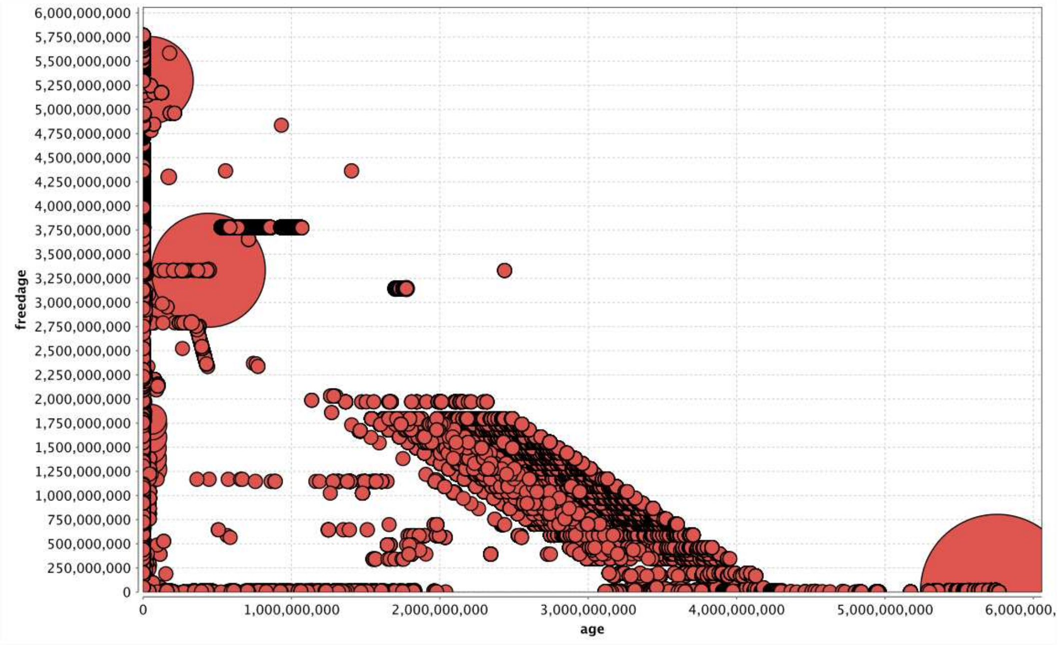

Figure 3.4: Tor: Age vs. Freed age bubble figure

3.1.2 Residual Object lifetime

Figures 3.2, 3.3 and 3.4 show how long freed objects last in memory before they are ultimately reallocated and clobbered. The residual lifetime, or **freed age** τ is a measure of this

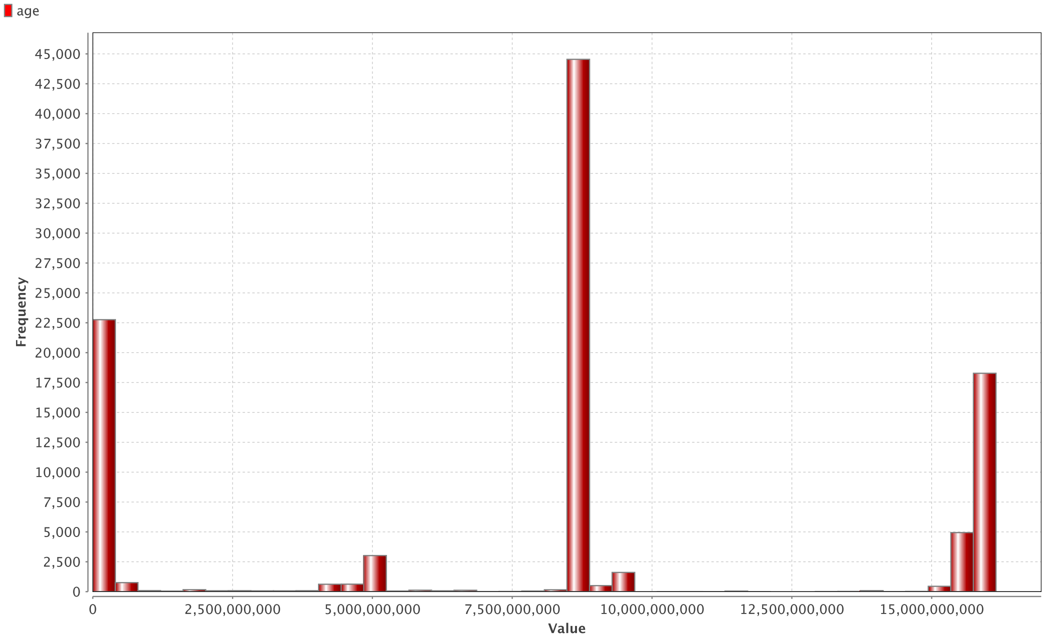

Figure 3.5: Firefox: Age histogram

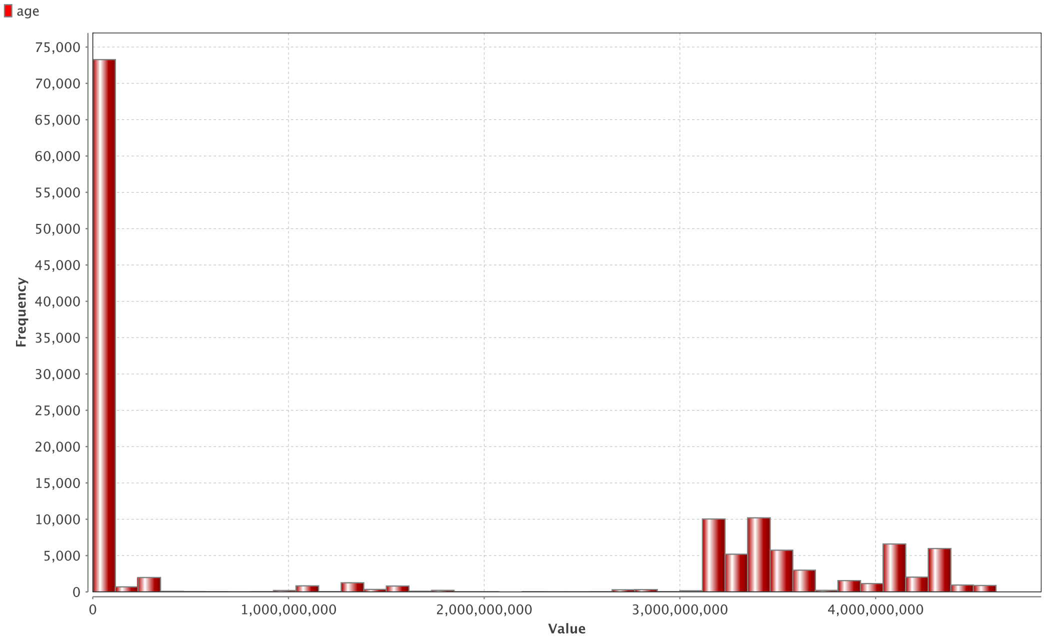

Figure 3.6: Konqueror: Age histogram

quantity and we define it as the length of time before a single byte of the freed object is clobbered. Although the residual age of an object may be short, if the allocation was large, it is likely to contain significant amounts of residual evidence, however, this evidence may

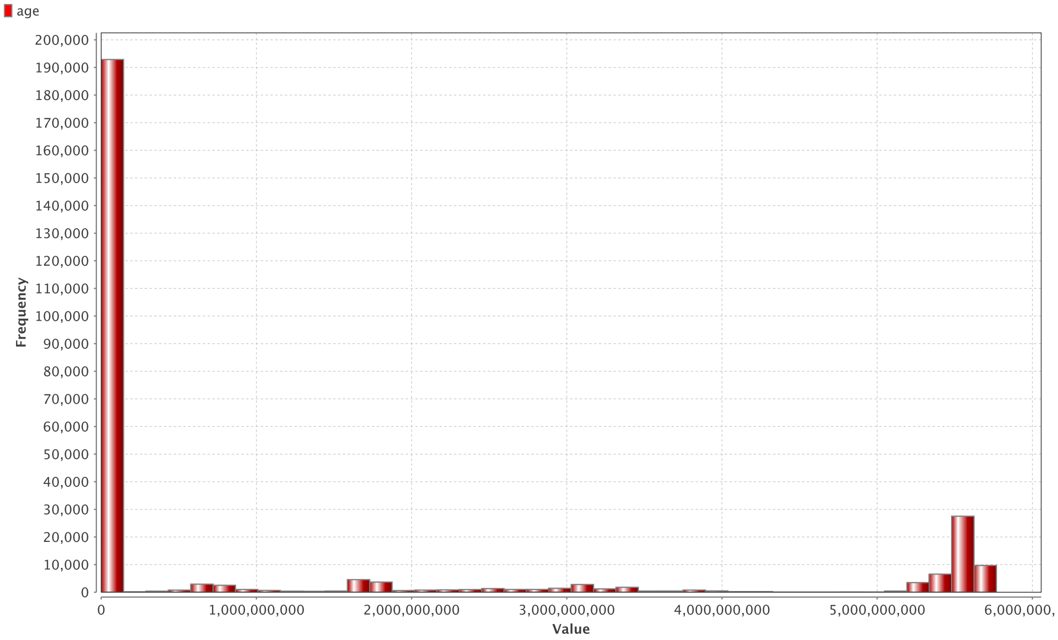

Figure 3.7: Tor: Age histogram

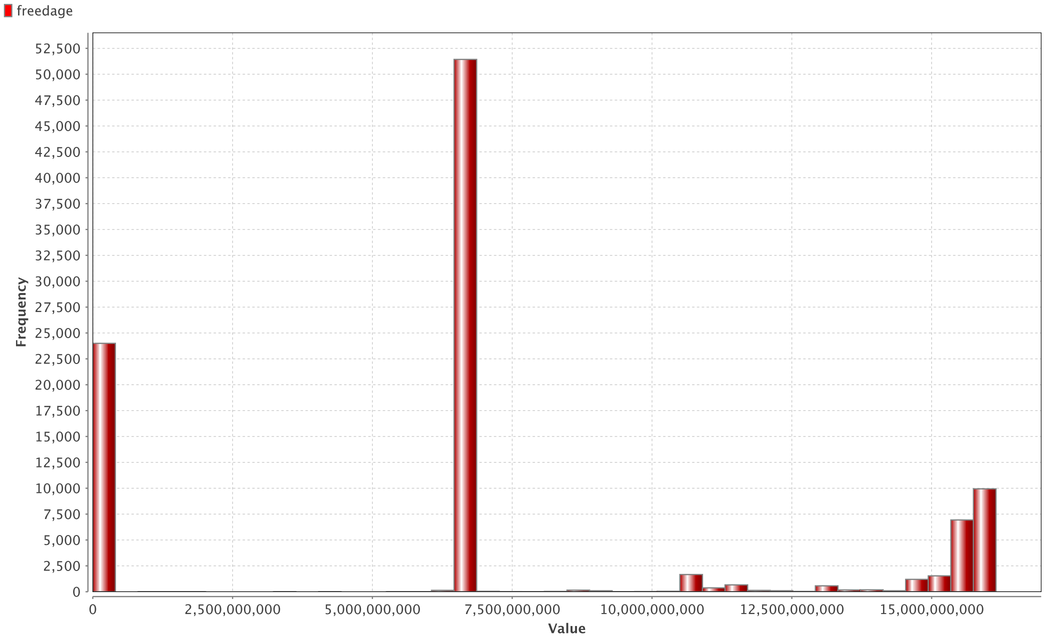

Figure 3.8: Firefox: Freed Age

be of low fidelity as it has been partially overwritten by new data.

Figure 3.2 shows a plot of freed age τ vs. age ω for Firefox in the unit of Valgrind virtual CPU cycles. The size of the bubble is proportional to the size of the allocation.

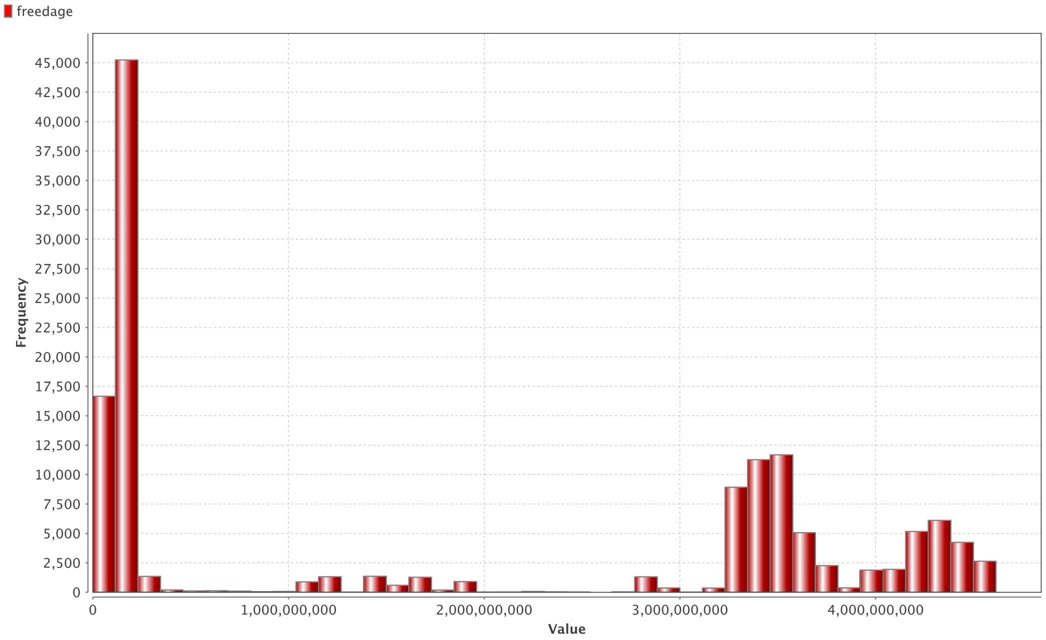

Figure 3.9: Konqueror: Freed Age

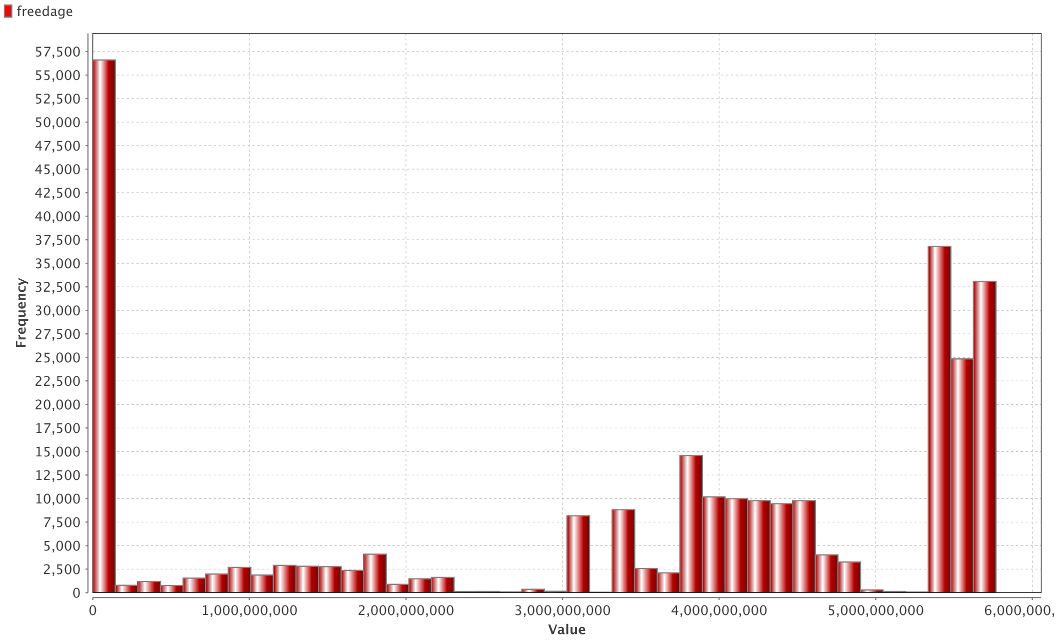

Figure 3.10: Tor: Freed Age

Recall that age is the natural lifetime of an object from allocation to deallocation and that freed age represents the amount of time that a residual object lasts before the memory area is reused. To the far right of the graph, we see objects with the maximal lifetime Ω. These objects persisted for the length of the session and they were deallocated because the program exited. Along the diagonal, objects have an age + freed age that represents an aggregate lifetime equal to the length of the program. Thus, $\omega + \tau = \Omega$ for the diagonal. We call this line the **key diagonal** where evidence lingers for the duration of the program lifetime. These objects were freed earlier, but their residual data persisted until the program terminated. Objects below the line have a shorter lifespan depending on the distance from the origin, as the available lifetime of the object is $\omega + \tau$.

From this bubble plot, we can see that there are some large allocations on the Y-axis that have a short natural lifetime, but long residual lifetime. This phenomenon is likely an artifact of the behavior of the memory allocator described earlier. Around the middle, a large number of objects are freed because Firefox uses a multi-stage teardown. First, pages/tabs are freed before the main UI exits. Figure 3.5 confirms this observation with three peaks. The first peak around zero shows that a good number of objects have very short lifetimes. The middle peak is when the tab/page gets torn down as part of the exit process and the final peak on the right is when the rest of the UI is torn down. The final plot of freed age shown in Figure 3.8 conveys similar information and reinforces the data shown in the other plots.

Similarly, Figure 3.3 shows a plot of the freed age vs. age for Konqueror. Unlike Firefox, this figure shows that large allocations tend to linger longer in Konqueror. We have not examined why this is the case; perhaps Firefox is more aggressive in cleaning up, or maybe Konqueror tends to reuse large objects. The large points near the origin imply that some large objects are allocated and deallocated rapidly, but their contents are overwritten fairly quickly. For instance, the RegEx parser exhibits this behavior. Likewise, objects along the key diagonal represent objects that persist until the program terminates.

Figures 3.6 and 3.9 depict histograms for Konqueror's age and freed age respectively. Not surprisingly, a large number of objects live on either end. Either the object is a temporary ephemeral allocation, or it lasts until the program exits. Unlike Firefox, Konqueror seems to clean up much more quickly at the end. Based on our observations, it appears that Firefox actually preserves data better than Konqueror because the latter tends to recycle large allocations more frequently. However, without knowing the exact details of how these programs work, it is hard to make generalizations about the actual longevity of residual evidence.

Finally, Tor is shown in Figure 3.4. Tor is interesting because large allocations tend to linger on the key diagonal, which is promising for evidence retrieval. Although some large allocations are freed quickly, their contents tend to linger for a long time afterward. Big objects on the key diagonal here are mostly large character buffers. Parallel diagonals in the figure reflect how Tor cleans up when it exits. Connections in Tor are circuit-based. When Tor exits, it shuts down one connection at a time. The act of doing so frees resources and the parallel diagonals illustrate how each connection shuts down. Each parallel line has the relationship: $age + freedage = shutdown\, time$. Tor's age histogram is relatively normal as depicted in Figure 3.7 with many short-lived allocations on the left, and some longer-term allocations on the right. However, its freed age shown in Figure 3.10 is reflective of the way that it manages its connections. These results are rather encouraging for forensic investigators and perhaps a bit shocking for privacy advocates. We have shown that a memory dump of a running Tor process does not necessarily scrub unused memory as aggressively as it can, otherwise the freed age would be unequivocally zero.

3.2 Characterizing Evidence

The data types discussed in Tables 3.2, 3.3 and 3.4 were identified using our evidence gathering techniques that highlight relevant data types based on content and role. We consider attributes such as ASCII text content, entropy level and pointer content. More details of these heuristics can be found in Appendix A.

By performing these data extraction experiments, we developed some rules of thumb for locating evidence. First, it appears that many applications use gzip to compress network streams. Finding gzip inflation buffers in memory appears to be a good strategy for recovering data, however, these buffers are frequently reused and their contents are unlikely to contain much historic data. Secondly, Firefox and Konqueror use smart pointers to manage objects. Looking for these smart pointers in memory can help highlight pointers to data. Third, applications tend to use container objects such as STL lists or tree nodes to manage important data. Locating key entry points to these internal structures generally leads to other interesting nodes. Finally, it seems like data is duplicated in many places. We found that after data has been decompressed, it is parsed in the XML layers and duplicates can appear in the caching and UI layers. Recognizing these patterns in the structure can help to identify information more accurately. Furthermore, these structures are often linked by pointers to each other.

More specifically, Table 3.2 shows data structures for Firefox. Not surprisingly, the objects with the longest lifetimes include the JavaScript hash table, garbage collector and the scoped XPCOM startup objects. Objects with the shortest lifetime include temporary helper objects such as parsers and decompressors. In terms of internal blurriness, objects with the highest write fraction include the garbage collector, which also has a high write velocity. Likewise, the decompressor has a write fraction of 34.05 and a write velocity of 300 million along with a short lifetime. This implies that the decompressor has a high turnover rate. Both Firefox and Tor use a data structure called `passwd` which is used by the `getpwnam()` function to return information about the currently active user. This data structure does not actually contain a web password.

Table 3.4 shows the statistics for Konqueror. Likewise, UI elements and settings persist the longest. Many objects in Konqueror are wrapped in smart pointers called DataRef(s) and these are managed by the internal reference counting framework. Since Konqueror makes extensive use of polymorphism using the QObject base class, analysis was significantly more time consuming because Cafegrind identified a less specific type. Like Firefox, Konqueror also makes compressed HTTP requests, but the write fraction was significantly lower at 13.55%. Perhaps the Firefox HTTP compression library is more integrated into the application whereas Konqueror seems to use gzip directly to decompress data.

Finally, Table 3.3 shows the statistics for Tor. Not surprisingly, most interesting structures in Tor involve encryption and their lifetime reflects the role of the object. For instance, bootstrap information such as the list of trusted directory services is initialized at the beginning. Individual ciphers are created as necessary for each connection and SHA state is created early on to authenticate the peer routing nodes. Expectedly, the SSL I/O buffer write ratio was relative high because only part of the buffer was filled and the rest was memset to 0.

Our findings confirm the observations of Chow's TaintBochs work [31] that significant amounts of evidence can persist in memory for quite a length of time. Our more recent study extends upon Chow's work by considering object types in the analysis instead of just raw taint bytes which are type agnostic. In contrast to TaintBochs, we do not track system-level taint because we are concentrating on measuring application-level behaviors. TaintBochs has already shown that the Linux kernel can persist sensitive data such as passwords and encryption keys in keyboard buffers, network ring buffers, flip buffers and the SLAB allocator used by the Linux kernel.

The empirical study of object lifetimes and modification rates here is meant only as a rough measure of these quantities. Modern programs are extremely complex and their behavior often relies on external input. Furthermore, measuring these quantities is known to be complicated because browsers such as Firefox use a COM-like interface[32] that relies on reference count-based garbage collection. Browsers generally use such an approach because page elements are shared between scripts, plugins and the page itself[28]. Therefore, these objects are only freed when the reference count drops to zero. Due to the involvement of multiple parties, these elements can persist for quite a length of time, but measuring the duration of this timeframe is very challenging and session-dependent.

Table 3.2: Firefox data structure lifetime and blurriness

Structure Name	Role	Lifetime(s) [%]	Write Fraction	Write velocity (x1000)
GCGraphBuilder	Garbage collector	31.76 [0.20]	87.29	20,083,945
GStringstr	Glib smart string	1436.00 [11.80]	26.90	30
JSScript	Javascript engine	4964.35 [41.10]	8.73	8.45
JSHashEntry	Hash table	7940.00 [65.70]	12.50	91
nsAttrValue	A struct that represents the value (type and data) of an attribute	4,629.00 [38.30]	17.08	8
nsCAutoString	Smart string	729.90 [6.04]	33.86	174
nsCOMPtr$< nsIContent >$	A node of content in a document's content model	5,470.00 [45.27]	25.36	16
nsCOMPtr$< nsICSSParser >$	CSS Parser	220.00 [1.80]	41.80	1583
nsCOMPtr$< nsIURI >$	Interface for a uniform resource identifier with i18n support	1171.00 [9.69]	27.85	9,992
nsEntry	Entry field	205.81 [1.70]	29.33	180
nsHTTPCompressConv	HTTP compression stream	54.72 [0]	34.05	300,000,000
nsXPTCVariant	XPT call helper for cross language/method calls	1045.00 [8.60]	24.80	27
passwd	Password from getpwnam()	0.81 [0]	26.05	562,809
png_struct	PNG image	1.16 [0]	31.40	1,319,082
Token	String tokenizer	938.00 [7.80]	16.40	548
ScopedXPCOMStartup	Manages XPCOM objects	9180.00 [75.90]	18.10	375
XML_ParserStruct	XML Parser	1.80 [0]	42.90	1,018,959

Table 3.3: Tor data structure lifetime and blurriness

Structure Name	Role	Lifetime(s) [%]	Write Fraction	Write velocity
(anonymous).cache_info	Cache information with info about clients	419.53 [34.80]	11.56	93.96
bio_f_buffer	OpenSSL buffered I/O	55.46 [0.05]	75.83	1510.06
dh_st	Diffie Hellman	4.25 [0.04]	29.25	156.52
rsa_st	RSA state	645.02 [53.50]	5.06	100.54
ssl_session	SSL session	160.22 [13.30]	100	353.81
trusted_dir_server	Trusted directory server	1205.38 [99.90]	64.40	18.73
SHAState	SHA state	921.69 [76.40]	33.33	14787.48
ssl_cipher_st	SSL cipher state	0.017 [0]	11.88	7613274.17

Table 3.4: Konqueror data structure lifetime and blurriness

Structure Name	Role	Lifetime (s) [%]	Write Fraction	Write velocity
DataRef$< type >$	Smart Pointer	2.49 [0.40]	18.73	20781.24
internal_state	Gzip state for compressed HTTP requests	82.05 [12.76]	13.55	5680172.31
KIconEngine	Icon engine	293.81 [45.69]	41.17	23.82
KUrl	URL	60.13 [9.35]	11.76	1157.80
passwd	Passwd from getpwnam()	0.12 [0]	26.05	3768948.12
png_struct	PNG image	0.15 [0]	9.50	559243.39
QLinkedListNode$< KHTMLPart* >$	List of embedded browser sessions	290.72 [45.21]	30.77	13.76
QVector$< type >$	Dynamic vector	6.57 [1.02]	30.25	6922.30

Chapter 4

Forenscope Overview

In contrast to traditional live forensic approaches which rely on running an agent on the host system, Forenscope utilizes the principle of introspection to provide a consistent analysis environment free of taint and blurriness which we term as the *golden state*[1]. In this state, the system is essentially quiescent and queries can be made to analyze the system. As a result, analysis modules can access in-memory data structures introspectively and query operating system services. Residual evidence described in Chapter 3 can be inspected and gathered with high fidelity.

The investigator activates Forenscope by forcing a reset where the state of the machine is preserved by memory remanence in the DRAM chips. Then, the investigator boots off the Forenscope media that performs forensic analysis on the latent state of the system and restores the functionality of the system for further live analysis. Forenscope is designed to work around security mechanisms by interposing a lightweight analysis platform beneath the operating system. For example, in incident response, the machine may be controlled by malicious software and the operating system cannot be trusted. The observation capabilities afforded by Forenscope offer additional visibility in these scenarios.

4.1 Memory Remanence

Modern memory chips are composed of capacitors that store binary values using charge states. Over time, these capacitors leak charge and must be refreshed periodically. To

[1] This state has no nodes on the untrusted path described in Section 2.2.1

save power, these chips are designed to retain their values as long as possible, especially in mobile devices such as laptops and cell phones. Contrary to common belief, the act of rebooting or shutting down a computer often does not completely clear the contents of memory. Link and May [33] were the first to show that current memory technology exhibited remanence properties back in 1979. More recently, Gutmann [34] elaborated on the properties of DRAM memory remanence. Halderman et al. [35] recently showed that these chips can retain their contents for tens of seconds at room temperature and the contents can persist for several minutes when the RAM chips are cooled by spraying an inverted can of commonly available computer duster spray over them. The liquid coolants found in these duster sprays freeze the chips and slow the natural rate of bit decay. Forenscope utilizes memory remanence properties to preserve the full system state to allow recovery to a point where introspection can be performed. We refer the reader to [35, 36, 37] for a more detailed analysis of memory remanence and the architecture of computer memory.

NB: The concept of memory remanence discussed in this section is distinctly different from the concept of residual data discussed in Chapter 3. The remanence here is an artifact of the physical properties of memory whereas the residual effect is purely an artifact of software design.

4.2 PC Memory

In this section, we will describe our study of post-boot up memory remanence characteristics on several X86-based systems. Even if a system's complete memory image is preserved at the instant that a computer is restarted, portions of the image are clobbered when the BIOS executes. When a PC is booted, the processor begins the initialization sequence by executing startup code from the BIOS. The BIOS is responsible for initializing the core hardware, performing any required selection of boot devices, and loading the boot loader or operating system from the selected boot device.

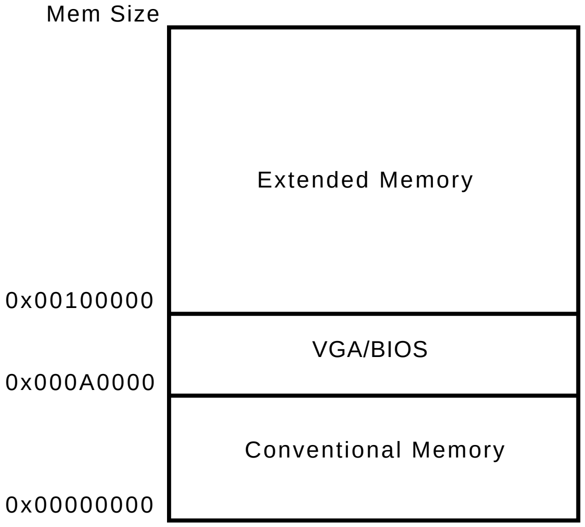

Figure 4.1: PC Memory map

Table 4.1: Regions of extended memory overwritten or not recoverable after BIOS execution

Computer	Type	Memory size	Overwritten or unrecoverable extended memory
HP/Compaq 8510w	Laptop	2 GB	1 MB @ 0x200000, 128 bytes @ 0x300000
HP/Compaq Presario 2800T	Laptop	392 MB	170 KB @ 0x100000
Lenovo Thinkpad T61p	Laptop	2 GB	3 MB @ 0x100000, 84 bytes @ various locations
IBM Thinkpad T41p	Laptop	2 GB	3 MB @ 0x100000, 84 bytes @ various locations
Dell Inspiron 600M	Laptop	512 MB	512 MB - completely reset
Dell Inspiron 8600	Laptop	1 GB	1 GB - completely reset
IBM IntelliStation M Pro	Desktop	512 MB	3 MB @ 0x100000, 856 bytes @ various locations (ECC off)
Bochs [38]	Emulator	128 MB	none
QEMU [39]	Emulator	128 MB	none

On a PC, there are two boot modes: cold and warm. A cold boot is performed when the machine is initially powered on. Cold boots perform significant hardware initialization and optional memory testing, which can overwrite portions of physical memory. Warm boots can be triggered by the reset button (if available) or by a reboot instruction from the running operating system. On a warm boot, there is no interruption in power and memory is usually left intact. In particular, a reboot initiated by the Linux kernel sets a CMOS flag that instructs the BIOS to bypass memory tests.

When booting, the processor starts running in X86 real mode and can initially address up to one-megabyte of memory. Only the first 640 kilobytes of addressable memory is RAM. The remainder of the one-megabyte addressable region is reserved for memory mapped I/O devices such as the display and other peripherals. This limitation is a consequence of the history and evolution of the PC architecture. Figure 4.1 shows the physical memory layout of a typical PC. The first 640 kilobytes of memory (up to 0x000A0000) is generally referred to as "Conventional Memory", while memory above the 1 MB mark is known as "Extended Memory." When the BIOS executes, it runs in conventional memory while it performs initialization and loads the boot sector from a disk.

On some machines, the system and peripheral BIOSes also access and overwrite select portions of extended memory. In order to perform a more detailed analysis of this issue, we have developed a RAM tester tool that analyzes the memory preservation characteristics of a computer. The tester is a stand-alone program that is loaded off a CD and uses only conventional memory to operate. The tester fills extended memory with a bit pattern, reboots the computer and checks for regions that have been overwritten or are unrecoverable. Table 4.1 presents the results of running our tool on several machines without ECC. These results show that various machines that differ in age and origin only overwrite a few megabytes of memory and can support memory preservation. The characteristic clobbering behavior of various machines is possibly a consequence of different BIOS implementations or hardware configuration discrepancies. We have not attempted to reverse engineer the

BIOSes on these machines to determine the exact reasons for these differences; however, we have noticed that on test machines, the absence or presence of various PCI expansion cards helps to determine the clobber signature of memory. Therefore, we believe that part of this effect can be attributed to the actions of expansion card firmware.

Our experiments indicate that machines which use ECC memory do not retain software-accessible memory contents after a restart. All reads from ECC memory after a reset always return 0 because ECC memory has parity bits that must be initialized by the BIOS at boot time [35]. Thus, computers that use ECC memory do not support memory remanence-based forensic techniques. We observed that disabling ECC functionality on ECC memory modules using BIOS settings makes these modules behave like non-ECC memory modules. The reader is referred to [35] for a more detailed analysis of memory remanence effects in DRAM chips.

Another important characteristic to consider is the behavior of the processor cache during a soft restart. This concern is significant in the context of a write-back cache, since uncommitted data may be lost across a reboot. Our experiments on hardware indicate that the cache is disabled by the processor and transparently recovered after a reset. We believe that when caching is re-enabled, the contents of the cache are preserved and remain intact for the final restoration to the original executing environment. When Forenscope re-enables virtual memory and caching, special care is taken to avoid clearing or invalidating cache lines.

4.3 Activation

Forenscope currently supports two methods of activation. The first is based on a watchdog timer reset and the second is through a forced reboot.

For incident response, or use with a honeypot, a watchdog timer may be used to activate Forenscope periodically to audit the machine's state and check for the presence of stealth malware threats. Watchdog timers are commonly used in embedded systems to detect erroneous conditions such as machine crashes or lockups. These timers contain a count down clock that must be refreshed periodically. If the system crashes, the watchdog software will fail to refresh the clock. Once the clock counts down to zero, the watchdog timer will promptly issue a hardware reset signal to the machine causing it to warm-reboot in the hopes that the operating system will recover from the erroneous condition upon a fresh start. On the SEL-1102 computer, the built-in watchdog timer is programmable via a serial port interface and the contents of DRAM memory are not cleared after a reboot initiated by the watchdog timer reset signal.

On the other hand, a forensic investigator may encounter a machine that is locked by a screensaver or login screen. In this situation, Forenscope can be activated by forcing a reboot. Some operating systems such as Linux and Windows can be configured to reboot or produce a crash dump by pressing a hotkey. These key sequences are often used for debugging and are enabled by default in many Linux distributions. In Linux, the *alt-sysrq-b* hotkey sequence forces an immediate reboot. If these debug keys are disabled, then a reset may be forced by activating the hardware reset switch. Forenscope supports multiple modes of operation for versatility. After the machine has been rebooted forcefully, the Forenscope kernel is booted instead of the incumbent operating system.

A successful attempt depends on the ability of the target computer to start from an external boot device. While this functionality can be disabled or password protected in the BIOS, many machines still have this ability enabled by default to allow the use of recovery CDs by technical support personnel. Even if the BIOS policy restricts booting from an alternate device, it is possible that a boot loader installed on the host system can be used to circumvent the system policy to load Forenscope from an external device if the loader isn't properly configured for security. All Linux distributions use a boot loader and we have verified that one of the most popular boot loaders, GRUB, can be instructed (at runtime) to boot from a different device irrespective of the BIOS boot policy.

Chapter 5

Forenscope Design

Instead of booting afresh, Forenscope alters the boot control flow to perform its analysis. Figure 5.1 illustrates this process. After the machine restarts, it boots off a CD or USB stick with the Forenscope media. The machine then enters the *golden state* monitor mode that suspends execution and provides a clean external view of the machine state. By booting into the golden state, Forenscope establishes a clean chain of trust(Chapter 2.2.1) so that no components of the target system provide critical services in the analysis process. To explain how the monitor works, we first describe the operating states of the X86 architecture. When a traditional PC boots, the processor starts in *real mode* and executes the BIOS. The BIOS then loads the boot loader that in turn loads the operating system. During the boot sequence, the operating system first enables `protected mode` to access memory above the 1 MB mark and then sets up page tables to enable virtual memory. This is the OS bootstrapping stage. Once the OS is booted, it starts scheduling programs and user-level applications. Forenscope interposes on this boot sequence and first establishes a bootstrap environment residing in the lower 640 KB rung of legacy conventional memory and then it reconstructs the state of the running machine. Forenscope has full control of the machine and its view is untainted by any configuration settings from the incumbent operating system because it uses a trustworthy private set of page tables; thus rootkits and malware that have infected the machine cannot interfere with operations in this state.

Next, Forenscope obtains forensically accurate memory dumps of the system and runs various kinds of analyses described later in this chapter. To maintain integrity, Forenscope does not rely on any services from the underlying operating system. Instead, it makes direct

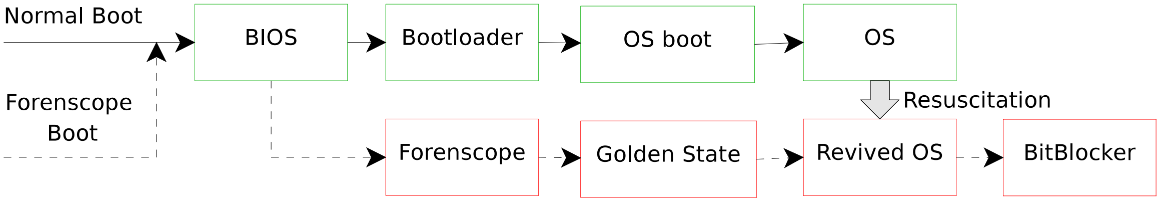

Figure 5.1: Boot control flow for normal and Forenscope boot paths

calls to the system's BIOS to read and write to the disk. Therefore, Forenscope is resistant to malware that impedes the correct operation of hardware devices. The initial forensic analysis modules are executed in this state and then Forenscope restores the operation of the incumbent operating system. This entire process is executed in a matter of seconds and the system is restored fully without delay to minimize disruption.

5.1 Reviving the Operating System

To revive the incumbent operating system, Forenscope needs to restore the hardware and software state of the system to ”undo” the effects of the reboot. Hardware devices are reset by the BIOS as part of the boot process. Some of these devices must be reconfigured before the incumbent operating system is restored because they were used by Forenscope or the BIOS during initialization. To do so, Forenscope first re-initializes core devices such as the hard drive and interrupt controller and then assumes full control of these devices for operation in its clean environment. Before resuming the operating system, Forenscope scans the PCI bus and gathers a list of hardware devices. Each hardware device is matched against an internal database and if an entry is found, Forenscope calls its own reinitialization function for the particular hardware device. If no reinitialization function is found, Forenscope looks up the device class and calls the operating system's generic recovery function for that device class. Many devices such as network cards and disk drives have facilities for handling errant conditions on buggy hardware. These devices typically have a *timeout re-*

covery function that can revive the hardware device in the event that it stops responding. We have found that calling these recovery functions is usually sufficient to recover most hardware devices. In Linux, 86 out of the 121 (71%) PCI network drivers implement this interface and all IDE device drivers support a complete device reset. For instance, the IBM uses an Intel Pro/100 card and the SEL-1102 uses a built-in AMD PCnet/32 chip. Both machines rely on calling the `tx_timeout` function to revive the network. We use a two-stage process to restore the operating system environment. The first stage reconstructs the processor state where the values of registers are extracted and altered to roll back the effects of the restart. Our algorithm scans the active kernel stack and symbol information from the kernel for call chain information as shown in Figure 5.2. Forenscope uses this information to reconstruct the processor's state. In the `alt-sysrq-b` case, the interrupt handler calls the keyboard handler that in turn invokes the emergency sysrq-handler. The processor's register state is saved on the stack and restored by using state recovery algorithms such as [36, 40]. More details are available in Section5.3. If the `alt-sysrq-b` hotkey is disabled, Forenscope supports an alternate method of activation based on pressing a physical reset switch. In this case, Forenscope assumes that the system is under light load and that the processor spends most of its time in the kernel's idle loop. In this loop, most kernels repeatedly call the X86 `HLT` instruction to put the processor to sleep. Since the register values at this point are very predictable, Forenscope simply restores the instruction pointer, `EIP`, to point to the idle loop itself and other registers accordingly. Once the state has been reconstructed, Forenscope reloads the processor with this information and enables the page tables to restore the state of virtual memory.

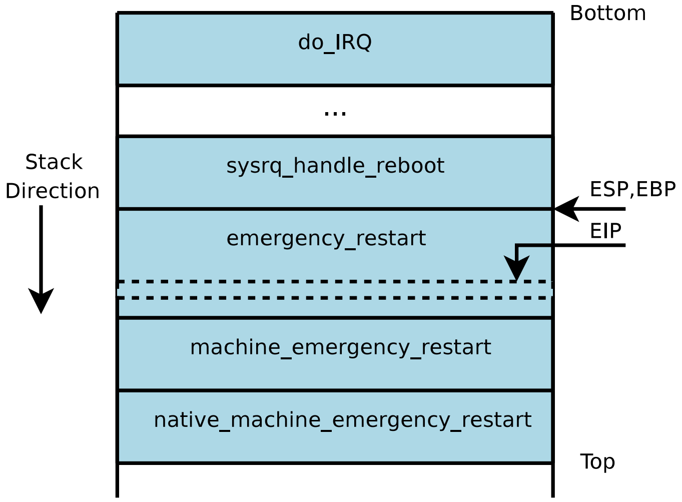

Figure 5.2: Stack layout at the time of reboot

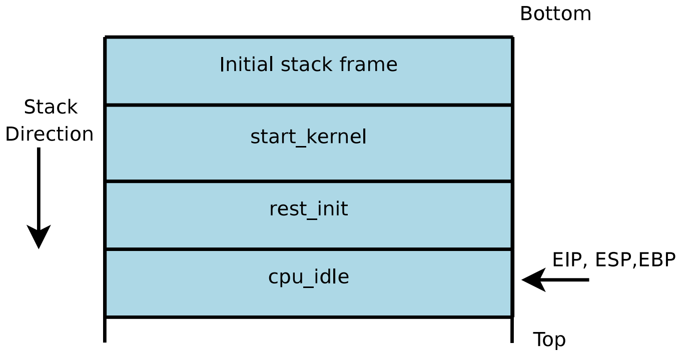

Figure 5.3: Idle stack layout at the time of reboot

5.2 Memory Map Characteristics

By running our RAM tester and analyzing the source code of Linux, we have determined that the contents of conventional memory have no operational effect on the system, even if the values are completely clobbered. This is because memory is not contiguous, and contains several holes around the 1 MB and 16 MB regions that are reserved for various devices such as expansion ROMs, legacy video cards, DMA regions and BIOSes. Since Linux primarily uses extended memory, any corrupted kernel regions in this memory presents a significant challenge for Forenscope.

We also performed some experiments to determine the extent to which Windows XP relies on conventional memory. In our experiments, we zeroed the contents of conventional memory from a Windows XP machine running in a VMware environment. In each case, Windows continued to run unaffected and DOS applications launched from the cmd.exe interpreter continued to work correctly. We believe that these experiments succeeded because Windows XP does not use conventional memory for DOS applications because it uses the Virtual Dos Machine (VDM)[41] to emulate DOS support and this environment actually uses extended memory to support the virtualized conventional memory. Dumping the contents of conventional memory also returned all zeroes in the Windows XP machine, which leads us to believe that conventional memory serves no operational purpose.

In some cases where portions of kernel code are overwritten, Forenscope can reload the kernel code section from the disk; however, when kernel data is corrupted, complete system recovery is non-trivial and we do not attempt to address this problem. Thus, our experiments assume that vital kernel data remains intact. Chapter 6 discusses strategies for dealing with such kernel data corruption.

5.3 System Resuscitation

Forenscope is designed to revive a system's software environment and hardware devices through resuscitation. After a machine is forcefully rebooted, the investigator reconfigures the target machine's BIOS, if necessary, to load Forenscope. Forenscope is loaded into conventional memory and only uses memory below the 640-kilobyte limit in order to avoid inflicting further damage to the preserved memory above 1 megabyte. Forenscope currently uses a two stage loading process. The first stage uses a slightly modified version of the GRUB boot loader [42] to load any required support files off the disk and the rest of Forenscope into memory. This modified boot loader is part of Forenscope and resides on the investigator's boot media; it should not be confused with the boot loader on the target

Table 5.1: Recovering register state

Register	Description	Source
EAX,EBX,ECX,EDX	General Purpose	Restored from stack
ESI,EDI	General Purpose	Restored from stack
ESP,EBP	Stack, Frame Pointer	SysRq handler stack or idle loop stack
EIP	Instruction Pointer	SysRq handler stack or idle loop stack
EFL	Flags	Restored from stack
CPL	Privilege Level	Kernel privilege level
CS,DS,SS	Code, Data, Stack Segments	Static value in kernel
ES,FS,GS	Segment Registers	Current task data structure
LDT,GDT	Descriptor Tables	Static value in kernel
TR	Task Pointer	Static value in kernel
CR0	Processor Settings	Static known value
CR2	Page Fault Address	Recovery not required
CR3	Page Table Pointer	Current task data structure
CR4	Advanced Control Flags	Static known value

machine. The second stage boot process executes the core system resuscitation code and forensic payloads.

5.4 Software Resuscitation

In this section, we describe the process of resuscitating a Linux environment on a target machine. Forenscope requires knowledge of kernel data structures and their layout (symbol table) in the target Linux kernel in order to successfully revive a system. For example, in order to restore the correct virtual memory mappings for Linux, Forenscope needs to discover the address of the page tables of the interrupted task. Such information is incorporated into Forenscope by building it with Linux kernel header files and using symbol information from the linked kernel. A compiled version of Forenscope is therefore only 100% effective against a specific kernel version and configuration. This is not a significant limitation because most major Linux distributions do not update kernels frequently and few people build and run custom kernels. Forenscope can generally accommodate changes in the kernel when it is recompiled; however, major Linux structural changes may require some additional engineering effort to support. In cases where the information is not directly available, Forenscope scans the system symbol file, /boot/System.map to find the necessary offsets to locate key kernel data structures. Since the format of these structures doesn't change often, Forenscope is able to cope with small changes.

The objective of software resuscitation is to fabricate a processor context (register state) that can be used to resume execution of the host system environment. We present a scheme that reconstructs a valid resume context for the Alt-SysRq-B reboot method (as discussed in Chapter 4). The key idea is to load the processor registers with values that cause the SysRq handler in Linux to mimic a return from the reboot function, as if it had no operational effect. This allows execution to continue in spite of an actual reboot. The advantage of using this approach to revive the kernel, as opposed to reviving the kernel from some other entry point, is that the logical control flow in the host system is preserved and the semantics of constructs such as locks and semaphores are respected.

In order to construct a valid resume context, it is necessary to understand the operation of the Linux Magic SysRq debug system. The SysRq system in Linux is hooked into the keyboard driver and called when the driver detects the Alt-SysRq hotkey sequence. When the SysRq handler is executed, it acquires a spin lock and disables interrupts. In this context, the system is guaranteed to have a very predictable register state and call chain. We rely on these salient characteristics to construct a resume context safely.

Table 5.1 lists the various X86 registers and describes the data sources used to reconstruct the values in each register. Forenscope first locates the stack that was in use at the time of restart. Depending on the kernel configuration, this can be one of several known stacks used by kernel threads and interrupt handlers. Figure 5.2 shows a glimpse of the stack contents just prior to restart. A number of known frames exist on this stack prior to executing the reboot code. These stack frames are scanned by Forenscope to recover register values to build a plausible resume context. Restore values for ESP and EBP are obtained by locating the frame for the *sysrq_handle_reboot()* function on the stack. Then, EIP is set to the return address of the *emergency_restart()* frame. Now, the execution context is prepared to synthesize a return from the reboot function as if the restart had been magically rolled back.

The *emergency_restart()* function does not have a return value and Forenscope does not have to worry about restoring any of the general-purpose registers or processor flags. Once the resume context is restored onto the processor, the return code in the function *sysrq_handle_reboot()* restores the values of these registers from the stack where they were saved in the function call preamble.

The values of the CS, DS and SS segment registers are constant within the kernel and these constants are statically compiled into Forenscope, while the values of the ES, FS, and GS registers are derived from the currently running task. Linux maintains a pointer to the data structure representing the currently running task in a global variable. Forenscope examines this variable and restores the ES, FS and GS registers with the values found in this data structure.

LDT and GDT always maintain a constant value independent of the executing thread. The task register, TR, holds relevant information about the current task. Linux uses software context switching, so there is only one TR per CPU, and its constant value is loaded by the kernel. All of these constant values are determined at build time and statically compiled into Forenscope. CR2 holds the fault address, which does not need to be recovered, and CR3 is the page table pointer, which can be inferred from the current task's virtual memory data structures. This information is required to activate the task's virtual memory mappings in the MMU. The virtual address space is constructed by consulting the page tables of CR3 and the values of the segment registers selected in the GDT and LDT tables.

Once Forenscope has constructed a valid resume context, it must load this context into the processor and enable the new page tables. This poses an interesting problem because

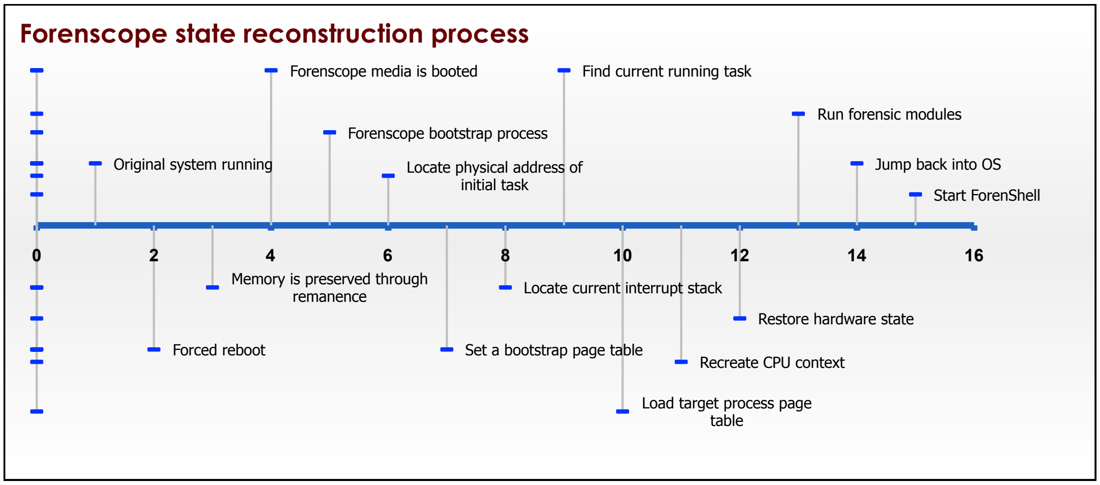
Forenscope state reconstruction process
Forenscope media is booted
Find current running task
Run forensic modules
Forenscope bootstrap process
Original system running
Locate physical address of initial task
Jump back into OS
Start ForenShell
0
2
4
6
8
10
12
14
16
Memory is preserved through remanence
Locate current interrupt stack
Restore hardware state
Forced reboot
Set a bootstrap page table
Recreate CPU context
Load target process page table

Figure 5.4: Forenscope state reconstruction algorithm

the code that reprograms the MMU to use the new set of page tables needs ensure that the next prefetched instruction is available in the old and new address spaces, otherwise the processor would generate an instruction fetch fault upon access. Unfortunately, Forenscope is loaded into conventional memory, which is not available in the standard kernel virtual memory map. Forenscope circumvents this problem by creating a common trampoline memory area that is available in both address spaces. It copies the code that completes the context restore process into an unused portion of the Linux kernel's log message buffer. When Forenscope returns to the host environment, it executes this code to atomically switch the virtual address space and return to the original host environment. Figure 5.4 depicts the state reconstruction process.

There are some limitations on the types of functions that can be called by Forenscope before it returns control to the kernel. Interrupts are disabled while Forenscope is running and it is essentially running in an atomic interrupt context; therefore, as with a real kernel interrupt handler, no functions that might sleep or otherwise invoke *schedule()* can be called. To work around this limitation, a standard Linux kernel API may be used to schedule an asynchronous callback to be run once interrupts are re-enabled; however, since Forenscope will no longer be mapped into memory, the actual machine code for the callback must be copied into an area accessible from the kernel address space in the same way that the trampoline code does. This technique is used by the ForenShell module discussed in Section 6.

5.5 Hardware Resuscitation

Upon reboot, system hardware may be reinitialized to a state that is out of sync with the expectations of the host Linux kernel. Forenscope needs to ensure that hardware devices such as keyboards, disk controllers and Ethernet devices continue to work after the software environment is revived. Surprisingly, Forenscope can rely on Linux to revive certain devices automatically. In order to handle some classes of buggy or unresponsive hardware, Linux includes code to retry operations or reinitialize devices upon an error. Forenscope takes advantage of such support if it is available. For cases where Linux is unable to revive hardware on its own, Forenscope incorporates custom code to assist resuscitation.

Forenscope's hardware resuscitation process is similar to the actions performed by the system when resuming from sleep or suspend mode. Normally, the operating system saves resume structure state to a disk or memory area upon a suspend. These resume structures describe the exact state of the machine at the time of suspend. Hardware devices and the CPU are then shut down and their transient state is subsequently lost. The process of rebooting incurs a similar loss of state. When the machine is resumed, the hardware and processor state are restored from a resume image. Forenscope does not have the luxury to save any state before the reboot and instead must fabricate an analogous restore state by scanning for clues in the residual memory image.

In this section, we describe the process of resuscitating the core hardware on a uniprocessor PC such as the keyboard, mouse, VGA graphics chip, interrupt controller, system timer and Ethernet chip. Most modern PCs have core hardware that is similar and backwards compatible with older revisions [43]. We describe restoration support for basic peripherals; this configuration is generally sufficient for system recovery. Each hardware device has its own peculiarities, and the recovery process is tailored to device semantics. We detail specific restoration challenges on a case-by-case basis.

Interrupt Controller:

X86 PCs use an i8259-compatible interrupt controller. This controller is responsible for arbitrating interrupts from various peripherals such as the keyboard, disk and network interface to the CPU. The i8259 controller supports interrupt masking and prioritization. On a PC, the highest priority interrupt is the system timer, followed by the keyboard. Other peripherals are usually assigned a dynamic interrupt number and priority by the PCI or plug and play (PNP) controller.

Upon reboot, the system BIOS completely resets the i8259 controller. All interrupts are masked and disabled. During recovery, we re-enable interrupts of active devices upon resuscitation. The i8259 also supports interrupt renumbering, which allows the controller to shift interrupt numbers by a base offset. Linux uses this facility to remap i8259 interrupts number 0-7 to Linux interrupts 32-39. Linux combines interrupts and processor exceptions (such as page faults) into a common linear exception table for fast lookup. Forenscope must restore this offset; otherwise Linux misinterprets the interrupt number read from the controller as a different processor interrupt, and causes a kernel panic by running the incorrect handler.

Some modern computers, especially those with multiple processors, use a newer interrupt controller called the Advanced Programmable Interrupt Controller (APIC). This controller is backward compatible with the i8259, and its semantics are similar. The code for resuscitating an APIC on a uniprocessor machine is similar to that of an i8259. At this time, we do not support multiprocessor machines.

System timer:

PCs have three i8253 programmable interrupt timers (PIT). Each timer can be configured for periodic or one-shot interrupts in accordance to a countdown value. The Linux kernel only uses the first timer as the system timer. The default setting for Linux is to configure a 100 Hz timer to periodically activate the scheduler. The other two timers are used by various drivers and applications.

Upon reboot, these timers are left intact. We do not need to reinitialize any of them. Although these timers do not need to be restored, their semantics are inconsistent when the system is resuscitated. For example, the system timer interrupt is not received by the kernel for the duration of the reboot, and hence the Linux *jiffies* software timer is suspended. Therefore, system time lags behind wall clock time. In order to preserve the semantics of software that relies on the correct time, we restore the system's time by reading the wall clock time from the hardware clock, which is always running (powered by a battery) even when the computer is turned off.

Starting with the Pentium, newer X86 processors include a monotonically increasing 64-bit CPU cycle time-stamp counter that is reset to zero whenever the processor is reset. The value of this counter can be read using the *rdtsc* instruction. Since this counter will be reset when the system is rebooted, code watching this counter for unexpected changes may be able to detect a discrepancy. In order to avoid detection, we estimate the value of the counter based on the system uptime and write this calculated value back to the counter using the *wrmsr* instruction.

PS/2 Keyboard/Mouse:

PS/2 keyboards and mice are driven by an i8042 controller. This controller is responsible for receiving keystrokes, mouse movements, and mouse clicks as well as setting keyboard LEDs. The controller itself is reinitialized by the BIOS at boot time. When GRUB runs, it initializes the controller and since Linux uses the same keyboard settings, no special re-initialization is required.

By default, the mouse is disabled by the BIOS at boot time. Forenscope sends the i8042 controller a command to re-enable the mouse port. Some PS/2 mice have settings for the scroll wheel and extra buttons. These settings are reinitialized by Forenscope to conform to the protocol of a standard scroll wheel mouse. Custom mouse protocols and advanced settings are vendor-dependent, and are thus not restored.

Display:

Although there are a myriad of video chips and drivers in the market, most chipsets support the standard VGA and VESA video modes. In Linux, the X server provides the graphics console and directly controls the hardware. When Forenscope reinitializes the device, it sets the video chip back to a safe state in text mode. Once the system is revived, the investigator can manually switch back to the X graphics console. The X server will perform the necessary re-initialization of the video device when switching back to the graphics mode.

Disk:

The IDE controller connects peripherals such as hard drives and optical disks to the system. Upon reboot, the BIOS reinitializes the disks and disables their corresponding interrupts on the i8259.

Forenscope eschews re-initialization of the disk controller, and instead relies on Linux's error recovery routines. IDE drives are accessed using a request/response command interface. If any command fails to respond due to disk or bus communication errors, Linux will retry the command or reinitialize the disk controller. This facility is used to deal with buggy IDE controllers and disks, which may hang under certain errant conditions.

When a disk is first accessed after Forenscope is run, there is an initial three-second lag while the initial IDE command times out and Linux resets the controller. Linux reports this disk error as a lost IDE interrupt, and transparently retries the command. Since the access is successful upon retry, no data read errors are reported, and the disk continues to operate smoothly without error.

Coprocessor units:

The Intel architecture has support for various coprocessors that are directly attached to the main CPU. One of the most popular units is the i387-compatible math coprocessor or floating-point unit (FPU). Since this coprocessor has become a fundamental part of the architecture, the kernel and user space C libraries expect it to be present and functional. Whenever the kernel performs a context switch, it saves the current process's FPU state and restores the FPU state of the process to be switched in. The user space GNU C library also initializes the floating-point unit during C program startup to ensure a predictable FPU state for C programs.

Upon reboot, the FPU is disabled by the system firmware. Forenscope must reset the FPU and re-enable it in order to clear any pending exceptions. Otherwise, the kernel's context switch code will fail when it suddenly discovers that the floating-point unit is off, and the FPU state cannot be saved or restored. The user space FPU settings are restored by the kernel's context switch code, so that applications do not mysteriously fail.

Network:

Upon reboot, the BIOS probes network devices and runs each device's respective firmware. This supports functionality such as network booting. We have found that many network chips are non-functional after resuscitation, so we must perform some re-initialization. Luckily, the Linux network driver model allows Ethernet devices to specify a transmit timeout function. This functionality allows a network interface to run a recovery routine if a transmission is held up in the send buffer, and does not get sent within a requisite timeout period. As part of the resuscitation process, Forenscope uses the Linux network device API to iterate through all network interfaces and calls each transmit timeout function. In many cases, this primes the network chip, and restores functionality. Since this API is part of the kernel's generic network device interface API, it does not depend on the card vendor, so many chipsets are supported without additional effort. We have confirmed that this approach works with the Intel PRO/100 and Intel PRO/1000 Ethernet adapters.

Typically, an Ethernet chip contains send and receive buffers. These buffers act as holding areas to queue outgoing packets when the Ethernet line is busy, and receive buffers to temporarily store unprocessed packets. Although Forenscope is able to recover the functionality of the Ethernet interface, there is the issue of packet loss while the computer is rebooting. These transient faults are always handled at higher network layers such as TCP and network connections do not usually experience any adverse affects. Since the reboot process requires less than half a minute, applications do not time out and subsequently close their connections.

5.5.1 Dealing with Diverse Hardware Configurations

Forenscope scans the PCI bus to determine the list of devices in the system. For each device, Forenscope checks to see if there is a specialized recovery routine for the device. If so, Forenscope executes the proper code to revive the hardware device.

If a specialized hardware recovery routine is not available, Forenscope calls the kernel's default recovery routine for the device. For instance, the network and hard drive device drivers both include timeout recovery routines in the event that the hardware fails to respond within a specified timeout.

In some cases, full recovery of the machine to the point where the graphical interface is restored is not necessary. Since restoring graphics may be non-trivial with certain chipsets, it may be a better strategy to simply obtain a command shell for analysis. With the shell, the investigator can change the user's password temporarily for the session (using the software write blocker to make the changes non-persistent) and login remotely to access the user's desktop environment. Furthermore, many kinds of analysis can be performed through the command-line and process-level memory dumps can also offer a wealth of helpful information.

Chapter 6

Modules

We have developed a number of modules to aid in forensic analysis. The modules are run in the order depicted in Figure 6.1. These modules run in groups where stage 1 modules run in the golden state to collect pristine information while stage 2 modules rely on OS services to provide a shell and block disk writes. Finally, stage 3 resumes the original operating environment.

Scribe:

Scribe collects basic investigation information such as the time, date, list of PCI devices, processor serial number and other hardware features. These details are stored along with the evidence collected by other modules to identify the source of a snapshot.

Cloner:

Cloner is a memory dump forensic tool that is able to capture a high-fidelity image of volatile memory contents to a trusted device. Existing techniques for creating physical memory dumps are limited by their reliance on system resources that are vulnerable to deception. Cloner works around forensic blurriness issues and rootkit cloaking by running in stage 1 before control is returned to the original host OS. In the golden state, the system uses `protected mode` to access memory directly through Forenscope's safe memory space. Using this technique, Cloner accesses memory directly without relying on services from the incumbent operating system or its page tables. To dump the contents of memory, Cloner writes to disk using BIOS calls instead of using an OS disk driver. This channel avoids a potentially booby-trapped or corrupted operating system disk driver and ensures that the written data has better forensic integrity. Most BIOS firmware supports read/write

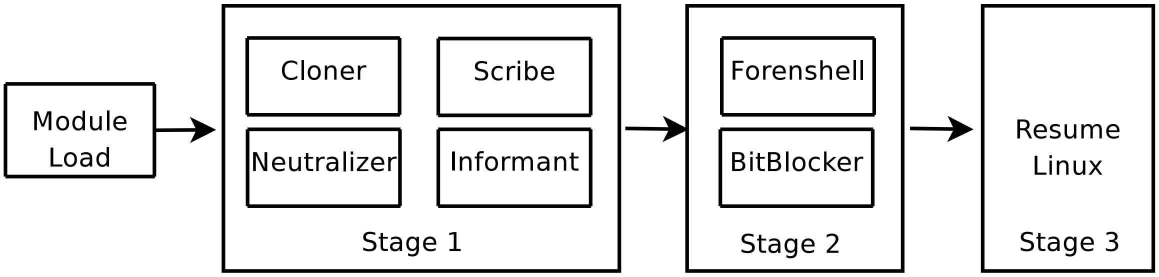

Figure 6.1: Forenscope modules

access to USB flash drives and hard disks. Another reason to use the BIOS for dumping is that it minimizes the memory footprint of Forenscope and reduces dependencies on drivers for various USB and SATA chipsets. Once Cloner captures a clean memory dump, the investigator can run other modules or tools that may alter the contents of memory without worry of tainting the evidence.

Memory dumps can be analyzed using offline tools such as Volatility [23] to find bits of residual evidence that may exist in the image. The encouraging results of Chapter 3 suggest that advances in forensic analysis techniques may one day help to uncover previously unreadable information.

Informant:

Informant checks for suspicious signs in the system that may indicate tampering by identifying the presence of alterations caused by malware. In order to extract clean copies of the program code and static structures such as the system call table, Forenscope must have access to a copy of the `vmlinux` kernel file that is scanned to locate global kernel variables and the location of various functions. Most Linux distributions provide this information. Read-only program code and data structures are checked against this information to ensure that they have not been altered or misconfigured. Such alterations have the potential to hinder the investigation process and Informant helps to assess the integrity of a machine

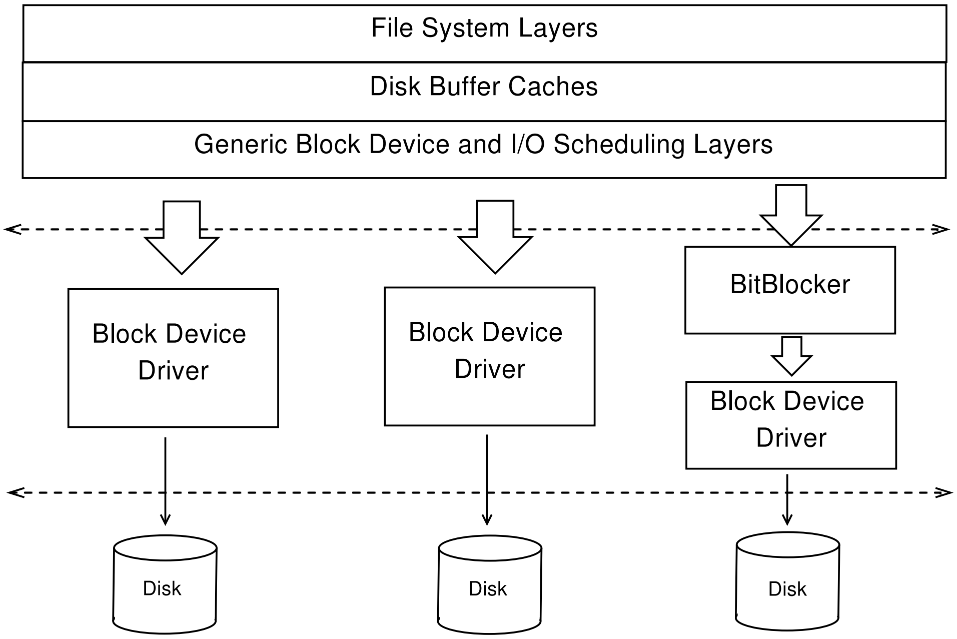

Figure 6.2: File system architecture

before further analysis is attempted. After Informant verifies the system, it also records other useful information such as the contents of the kernel *dmesg* log, kernel version, running processes, open files and open network sockets. This information can help expedite the investigation process by capturing a part of the system's dynamic state.

Neutralizer:

Neutralizer inoculates the system of anti-forensic software by detecting and repairing alterations in binary code and key system data structures such as the system call table. These structures can be repaired by restoring them with clean copies extracted from the original sources. Since many rootkits rely on alteration techniques, Neutralizer can recover from the effects of common forms of corruption. Presently, Neutralizer is unable to recover from corruption or alteration of dynamic data structures. Neutralizer also suppresses certain security services such as the screensaver, keyboard lock and potential malware or anti-forensic tools by terminating them. To terminate processes, neutralizer sends a `SIGKILL` signal instead of a `SIGTERM` signal so that there is no opportunity to ignore the signal. Customized signals can be sent to each target process. For some system services that respawn, terminating them is ineffective, so forcefully changing the process state to zombie (Z) or uninterruptible disk sleep (D) is desired instead of killing the application directly. An alternative would be to send the `SIGSEGV` signal to certain applications to mimic the effects of a crash. Neutralizer selects processes to kill based on the analysis mode. For incident response on server machines, a white list approach is used to terminate processes that do not belong to the set of core services. This policy prevents running unauthorized applications that may cause harm to the system. For investigation, Neutralizer takes a black list approach and kills off known malicious processes. Information to populate this blacklist can be found by consulting known malware databases that record the name and hash of known rootkits and bots. For certain domains such as industrial control and critical infrastructure, the whitelist of applications can be populated by listing only critical control processes and loggers. This approach has been shown to be fairly effective for embedded controllers running commodity operating systems.

ForenShell:

ForenShell is a special superuser *bash* shell that allows interactive exploration of a system by using standard tools. When coupled with BitBlocker(below), ForenShell provides a safe environment to perform customized live analyses. In this mode, ForenShell becomes nonpersistent and it does not taint the contents of storage devices. Once ForenShell is started, traditional tools such as Tripwire or Encase may be run directly for further analysis.

Forenscope launches the superuser shell on a virtual console by directly spawning it from a privileged kernel thread. ForenShell runs as the last analysis module after Informant and Neutralizer have been executed. At this point, the system has already been scanned for malware and anti-forensic software. If Neutralizer is unable to clean an infection, it displays a message informing the investigator that the output of ForenShell may be unreliable due to possible system corruption.

BitBlocker:

BitBlocker is a software-based write blocker that inhibits writing to storage devices to avoid tainting the contents of persistent media. Since actions performed by ForenShell during exploration can inadvertently leave undesired tracks, BitBlocker helps to provide a safe non-persistent analysis environment that emulates disk writes without physically altering the contents of the media.

Simply re-mounting a disk in read-only mode to prevent writing may cause some applications to fail because they may need to create temporary files and expect open files to remain writable. Typically, when an application creates or writes files, the changes are not immediately flushed to disk and they are held in the disk's buffer cache until the system can flush the changes. The buffer cache manages intermediate disk operations and services subsequent read requests with pending writes from the disk buffer when possible. BitBlocker mimics the expected file semantics of the original system by reconfiguring the kernel's disk buffer cache layer to hold all writes instead of flushing them to disk. This approach works on any type of file system because it operates directly on the disk buffer that is one layer below the file system. BitBlocker's design is similar to that of some Linux-based RAM disk systems [44] which cleverly use the disk buffer as a storage system by configuring the storage device with a null backing store instead of using a physical disk. Each time a disk write is issued, barring a *sync* operation, the operating system's disk buffer subsystem holds the request in the buffer until a certain write threshold or timeout is reached. In Linux, a system daemon called *pdflush* handles flushing buffered writes to disk. To prevent flushing to the disk, BitBlocker reconfigures the write threshold of the disk to inhibit buffer flushing, disables *pdflush* and hooks the *sync*, *sync_file_range*, *fsync*, *bdflush* and *umount* system calls with a write monitor wrapper. Figure 6.2 shows the architectural diagram of the Linux filesystem layer and where BitBlocker intercepts disk write operations. Although BitBlocker inserts hooks into the operating system, it does not interfere with the operations of Informant and Neutralizer because those modules are run before BitBlocker and they operate on a clean copy of memory. The hooks and techniques used by BitBlocker are common to Linux 2.6.x kernels and they are robust to changes in the kernel version. Similar techniques should be possible with other operating systems.

NetLogger

Since the contents of network connections established through encrypted tunnels such as VPNs and IPSEC cannot be directly analyzed on the physical wiring, it is challenging to get a good packet trace of such traffic using conventional tools. NetLogger relies on host tools such as TCPdump or Ethereal to generate plaintext traces of sessions originating from the host. This can be helpful if the machine being analyzed is a VPN concentrator or router that manages many connections to the internal network.

Reincarnator:

Reincarnator restores a memory dump to a physical system. Instead of relying on memory remanence, Reincarnator reads a raw memory dump from a storage device back to memory and then runs Forenscope's state reconstruction algorithm to resuscitate the system. This module should be used with care since it assumes that the underlying system state has not changed. For instance, if a memory dump is taken with Cloner and restored with Reincarnator, the contents of the underlying disk should match the contents at the time of state capture.

A forensic investigator can use this module as follows. First, the investigator acquires a consistent memory dump using Cloner. Then, the original hard disk is copied for forensic soundness. Once a copy is made, the investigator can then install the cloned disk into the original machine and run Reincarnator to restore the machine. Care should be taken to ensure that the replacement disk has the *e*xact configuration as the original disk; otherwise corruption may occur due to using a different hardware driver or due to variations in the low-level disk block layout.

If the machine is using standard IDE drivers and a generic set of drivers (PC-compatible), then it may be possible to restore to a virtual machine. We have tested this to a limited capacity in QEMU and the results seem promising. This suggests that error recovery facilities in operating systems are robust enough to support advanced recovery even if the hardware is completely reset or reinitialized.

Chapter 7

Results and Evaluation

We evaluate Forenscope against the requirements of a good forensic tool and measure five characteristics: correctness, performance, downtime, fidelity and effectiveness against malware.

7.1 Hardware and Software Setup

To demonstrate functionality, we tested and evaluated the performance of Forenscope on two machines: a SEL-1102 industrial computer and an IBM Intellistation M Pro. The SEL-1102 used in our experiments is a rugged computer designed for power system substation use and it is equipped with 512 MB of DRAM and a 4 GB compact flash card mounted in the first drive slot as the system disk. The SEL-1102 can operate in temperatures ranging from -40 to +75 degrees Celsius. The IBM Intellistation M Pro is a standard desktop workstation equipped with 1 GB of DRAM. For some tests, we opted to use a QEMU-based virtual machine system to precisely measure timing and taint. Forenscope and the modules that we developed were tested on the Linux 2.6 kernel. Although Forenscope was originally built to target Linux, we plan to expand this work to other systems.

7.2 Correctness

We tested Forenscope against a mix of applications to show that a wide range of hardware, software and network applications are compatible. Correctness is assessed by comparing the output of several applications in a control environment against corresponding output in a test environment. Effectiveness against security programs is demonstrated by bypassing several encryption utilities. In each case, Forenscope is able to recover the operating state of the system without breaking the semantics of running applications.

Please note that the actual encryption algorithms are not understood by Forenscope, and that the security mechanism bypassed here is the screensaver or console lock. These tests merely demonstrate that the security applications are not sensitive to a mediated reboot.

To confirm that Forenscope is able to correctly resuscitate the system without corrupting memory or the processor cache, we run a standard memory tester called *memtester* [45], and reboot the machine during several test runs. The tests we selected create patterns in two separate memory areas by either writing random values or performing some simple computations such as multiplication or division. It then compares the two areas to ensure that no memory errors have occurred. We ran over 10 trials of these tests on our evaluation system while restarting the system in the middle of the test. In all of these cases, memtester does not report any errors.

In order to verify the correctness of the system after resuscitation, we force a restart in the middle of several different tasks and examine if the tasks complete correctly after the resuscitation.

The tasks performed are:

1. *gcc*: Compilation of the C source file containing the H.264/MPEG-4 AVC video compression codec in the MPlayer [46] media program.
2. *gzip*: File compression using the deflate compression algorithm.
3. *wget*: File download.

4. *convert*: JPEG image encoding.
5. *aespipe*: AES file encryption.

For the last four tasks, we use a 100 MB TIFF image file as the input. For each task, we compare MD5 hashes of the output file generated without a Forenscope interruption against results obtained from a test with a restart in the middle of task execution. We have run each experiment at least 10 times and have not encountered any errors.

The primary goal of Forenscope is to provide the investigator with access to protected data on a live system transparently. In order to demonstrate that this goal has been attained, we selectively test several applications and kernel-level subsystems that provide secure access to data and network resources. All of these programs store temporary keys in volatile memory.

The applications we test are:

1. *SSH* secure shell connection between the target and another host.
2. *SSL* web browser session to a secure web server.
3. *PPTP* [47] VPN connection to a secure university network.
4. *dm-crypt* [48] encrypted file system.
5. *Loop-AES* [49] encrypted file system.

In the first two cases, the secret keys are located in user space. In the remaining cases, the keys are stored in kernel memory. We perform each test on a machine that is locked by a user (using a console locking program) with these programs running in the background. In each case, the Neutralizer module was used to terminate the lock program after resuscitation.

Our SSH tests use OpenSSH version 4.6 as the client and server. OpenSSH supports various modes of the AES, DES, Blowfish and ARC4 ciphers. In each case, Forenscope was able to allow an investigator to access the secure shell irrespective of the underlying encryption scheme or key size that was used. The SSL test consists of a web browser session connected to a popular web e-mail service using TLS 1.0 and AES encryption. After restarting, we were still able to browse mail folders, read messages, and send e-mail using the seized session. The VPN client we used was based on the Point-to-Point Tunneling Protocol (PPTP) [47] with 128 bit keys. VPN connections and application sessions remained operable after a successful run, and new connections could be established over the open tunnel. This implies that enterprise level perimeter defenses can be successfully overcome using Forenscope. We also tested Forenscope against two popular Linux file system encryption systems: dm-crypt and Loop-AES. Both systems were configured to use 256-bit AES CBC encryption. We were able to access the contents of the file system successfully.

These experiments demonstrate the significance of our approach. Other published key retrieval techniques [35, 50, 51, 52, 53, 54] must be tailored to specific key types, sizes, applications or kernel components. Such techniques require extra support to use the retrieved keys to gain access to secure sessions. Forenscope allows the investigator to access the live system and immediately explore or manipulate protected data.

To evaluate the correctness of BitBlocker, we ran it on the IBM and on a QEMU system emulator. Using the emulator allowed us to verify integrity by checksumming the contents of the virtual disk. Our test cases include using the *dd* utility to fill up the disk, then issuing a *sync* command and unmounting the disk. Other cases tested include copying large files and compiling large programs consisting of hundreds of files. In each case, BitBlocker worked correctly and no writes were issued to the physical disk. After the test completed, we confirmed that the contents of the disk were unchanged by comparing hashes of the contents against the original contents.

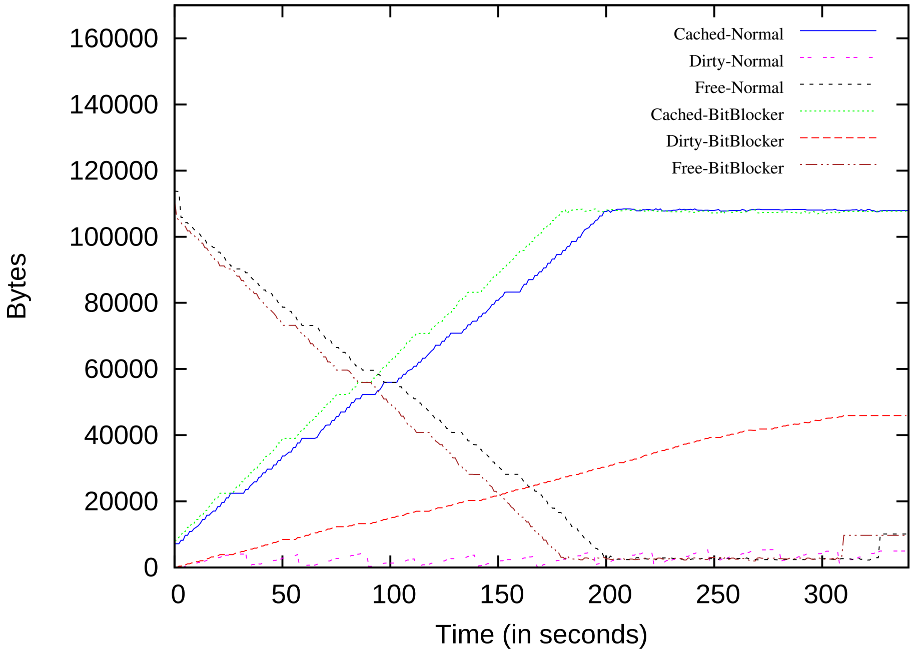

Figure 7.1: BitBlocker memory usage

7.3 Performance

Forenscope loads and revives the operating system within seconds. An interesting side effect is that BitBlocker makes disk operations appear to be faster because no data is flushed to the physical disk from the disk buffer. A write of a 128 MB file took 32.78 s without BitBlocker and 3.71 s with BitBlocker. The number of dirty disk buffers consumed increases proportionately with the size of the files written. Since BitBlocker inhibits flushing to disk, running out of file buffers can create a condition where the filesystem fills up and reports a write error. To measure these effects on the system, we collected buffer cache usage information once a second in several key applications: creating a compressed archive with *tar-bzip2*, downloading a file using *wget* and compiling the software package *busybox*. Figure 7.1 shows the utilization of dirty file buffers over time for the tar-gzip case. Wget and busybox compilation have similar results. In the graphs, we report statistics from `/proc/meminfo` such as `cached`, `dirty` and `free`. According to the documentation for `/proc`, `cached` in Linux represents the amount of data in the page cache which in-

Table 7.1: Linux memory measurements (borrowed from the /proc man page)

Quantity	Description
MemFree	The amount of physical RAM, in kilobytes, left unused by the system
Buffers	The amount of physical RAM, in kilobytes, used for file buffers
Cached	The amount of physical RAM, in kilobytes, used as cache memory
Active	The total amount of buffer or page cache memory, in kilobytes, that is in active use. This is memory that has been recently used and is usually not reclaimed for other purposes
Dirty	The total amount of memory, in kilobytes, waiting to be written back to the disk
Writeback	The total amount of memory, in kilobytes, actively being written back to the disk

cludes cached data from read-only files as well as write buffers. `Dirty` represents items that need to be committed to the disk and free represents free memory. Table 7.1 summarizes these measurements.

From our observations, `dirty` is generally very low in the normal case because the kernel commits write buffers periodically. However, in BitBlocker, `dirty` grows steadily because the data cannot be committed back to the disk. To estimate the amount of memory required to run BitBlocker, our experiments show that in many scenarios, even 128 MB of free memory is sufficient for BitBlocker to operate. Our experiments show that BitBlocker is robust even when the system runs low in memory. At 200 seconds, the physical memory of the machine fills up and the tar-bz2 process stops because the disk is ”full.” The system does not crash and other apps continue to run as long as they do not write to the disk. On a typical system with 2 GB of memory, BitBlocker should be able to maintain disk writability for a much longer period of time.

Downtime:

As discussed earlier, one important metric for evaluating a forensic tool is the amount of downtime incurred during use. To show that Forenscope minimally disrupts the operation of systems, we measured the amount of time required to activate the system. Forenscope

Table 7.2: Forenscope execution times

Task	Time (s)
Forced reboot on SEL-1102	15.1
Forced reboot on IBM Intellistation	9.8
Watchdog on SEL-1102	15.2

executed in 15.1 s using the reboot method on the SEL-1102 and in 9.8 s on the IBM Intellistation while the watchdog method took 15.2 s to execute on the SEL-1102. Table 7.2 summarizes these results. The majority of the downtime is due to the BIOS boot up sequence and this downtime can be reduced on some machines. Many network protocols and systems can handle this brief interruption gracefully without causing significant problems. We tested this functionality by verifying that VPN, SSH and web browser sessions continue to work without timing out despite the interruption. Many of these protocols have a timeout tolerance that is sufficiently long to avoid disconnections while Forenscope is operating and TCP is designed to retransmit lost packets during this short interruption.

To measure the disruption to network applications caused by running Forenscope continuously over a period of time, we ran a test designed to mimic the brief reboot cycle used by the analysis process. The test measures the instantaneous speed of an HTTP file transfer between a server and a client machine. While the file transfer is in session, we periodically interrupt the transfer by forcibly restarting the machine and subsequently reviving it using Forenscope. Each time the system is interrupted, the server process is suspended while the machine reboots. The process is then resumed once Forenscope is done running. As a baseline, we created a control experiment where the server process is periodically suspended and resumed by a shell script acting as a governor to limit the rate at which the server operates. This script sends the `SIGSTOP` signal to suspend the server process, waits a few seconds to emulate the time required for the boot up process and then sends a `SIGCONT` signal to resume operation. In each experiment, a `curl` client fetches a 1 MB file from a `thttpd` server at a rate of 10 KB/s. We chose these parameters to illustrate

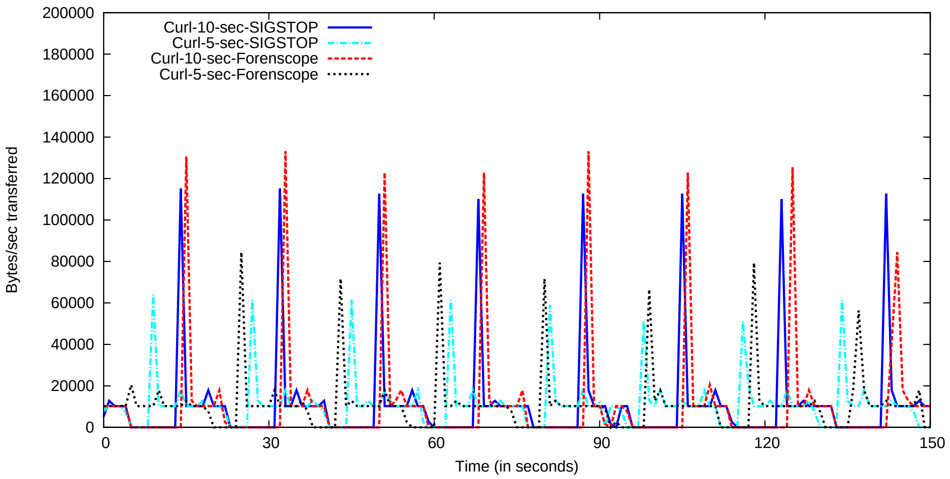

Figure 7.2: HTTP data transfer rate comparison

how a streaming application or low-bandwidth application such as a logger may behave. During this download process, the server was rebooted once every 20 seconds and we measured the instantaneous bandwidth with a boot up delay of 5 and 10 seconds to observe the effects of various boot up times. We observed that the bandwidth drops to zero while the system boots and the download resumes promptly after the reboot. No TCP connections were broken during the experiment and the checksum of the downloaded file matched that of the original file on the server. A graph of the instantaneous bandwidth vs. time is plotted in Figure 7.2. We compared the results of our test against the control experiment and observed that the behavior was very similar. Thus we believe that running Forenscope can be considered as safe as suspending and resuming the process. During the experiment we noticed that the bandwidth spiked immediately after the machine recovered and attribute this behavior to the internal 2-second periodic timer used by *thttpd* to adjust the rate limiting throttle table.

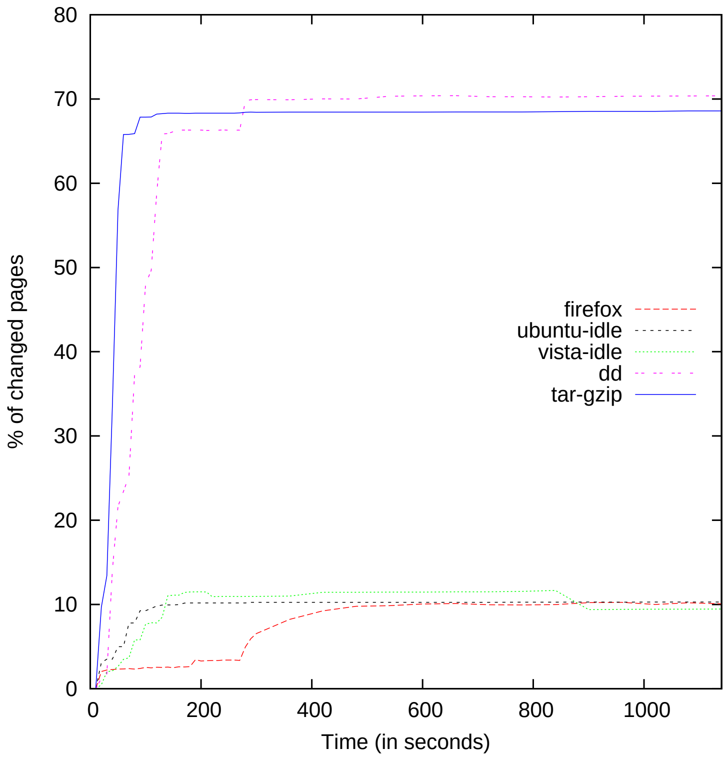

Figure 7.3: Comparison of memory blurriness

7.4 Taint and Blurriness

We evaluated the taint in a snapshot saved by Forenscope using a snapshot captured by `dd` as the baseline. In an experimental setup running with 128 MB of memory, we collected an accurate snapshot $\overline{S_t}$ of the physical memory using QEMU and compared that with a snapshot $\hat{S}_v$ obtained from each forensic tool. The number of altered pages for each of the configurations is presented in Table 7.3. We observe that since Forenscope is loaded in conventional memory, the only pages that differ are found in the lower 640 KB of memory. Our experiments show that Forenscope is far better than `dd` because no difference was observed in the extended memory between the snapshot taken by Forenscope and the baseline snapshot. It should be noted that as the machine is suspended in the golden state when running Forenscope, there is no blurriness associated with the snapshot taken by Forenscope. For *dd*, we measured the taint when using a file system mounted with and without the *sync* option. The number of pages affected remains almost the same in both cases and we observed that the majority of second-order taint was due to the operating system filling the page-cache buffer while writing the snapshot. To evaluate how much taint was induced due to buffering, we ran experiments in which *dd* was configured to write directly to disk, skipping any page-cache buffers by using the `O_DIRECT` *open()* flag. The results show that the taint was much lower than the earlier experiment, but still greater than the taint caused by using Forenscope. In order to estimate the amount of blurriness caused when tools like *dd* are used, we measured the natural drift over time of some typical configurations. We collected and compared memory dumps of systems running Ubuntu 8.04 with 512 MB of memory in a virtual machine environment hosted in `QEMU`. In each case, we snapshot the physical memory of the virtual machine and calculate the number of pages that differ from the initial image over a period of time. The snapshots were sampled using a *tilted time frame* to capture the steady state behavior of the system in an attempt to measure δ_v. The samples were taken at 10-second intervals for the first five minutes and at 1-minute intervals for the next two hours. From Figure 7.3, we observe that the drift remains nearly constant after a short period of time for our experimental setup and for the idle Ubuntu and Vista systems, the drift stabilizes within a few minutes. The drift for a system running Mozilla Firefox was found to be nearly constant within 10 minutes. Running *tar* and *gzip* for compressing a large folder or *dd* to dump the contents of memory into a file resulted in most of the memory being changed within a minute due to second-order taint. To summarize, our tests demonstrated that there is no taint introduced in the extended memory by using Forenscope and that Forenscope can be used for forensic analysis where taint needs to be minimized. Forenscope compresses data to maximize the efficiency of capturing a memory image. Common compression tools, such as gzip or bzip2, require more code and scratch memory than there is available within the allotted 640 KB of conventional memory. We use a lightweight library called *quicklz* because it resides in 2.4 KB of code, uses 46 KB of memory for scratch space, and provides average compression ratios around 47.5%.

Description (Total of 32768 pages)	Pages changed in Conventional Memory	Pages changed in Extended Memory
Forenscope	41 (0.125%)	0(0%)
dd	0 (0%)	7100 (21.66%)
dd to FS mounted with sync flag	0 (0%)	7027 (21.44%)
dd with O_DIRECT	0 (0%)	480 (1.46%)

Table 7.3: Taint measurement

7.5 Trusted Baseline

We compared the output of Forenscope's process list against that reported by *ps* on a pristine system and found that the results agreed with the control experiment. Similar experiments were performed against the output of *netstat*, and *lsof* that lists open files on the system. Memory dumps were tested by performing a dump of memory from */dev/mem* and comparing against a Forenscope memory dump when the system was idle. Due to the issue of forensic blurriness, there were some artifacts that were reported differently between the memory dumps, but special markers we inserted into the memory were correctly identified along with crucial portions of the kernel. To further demonstrate correctness, we performed a memory dump with the QEMU emulator and compared it against the dump taken by Forenscope. The memory images matched with the exception of the interrupt stack because QEMU was able to halt the machine during the dump phase but the process of running Forenscope triggered the execution of *sysrq* handling code. Therefore, we are confident that our tool is no worse than existing dumpers, and operationally no worse than a baseline memory dump from an emulator.

7.6 Effectiveness Against Anti-forensic Tools

Although forensics techniques can collect significant amounts of information, investigators must be careful to ensure the veracity and fidelity of the evidence collected because anti-forensic techniques can hide or intentionally obfuscate information gathered. In particular, rootkits can be used by hackers to hide the presence of malicious software such as bots running in the system. Malware tools such as the FU rootkit[55] directly manipulate kernel objects and corrupt process lists in ways that many conventional tools cannot detect. Malware researchers have also demonstrated techniques to evade traditional memory analysis through the use of low-level rootkits[56] which cloak themselves by deceiving OS-based memory acquisition channels on Linux and Windows. Hardware [57] and software [58] virtualization-based rootkits may be challenging to detect or remove by the legitimate operating system or application software because they operate one layer below standard anti-malware facilities. We describe and evaluate how Forenscope reacts to several publicly available rootkits. The set of rootkits was chosen to cover a representative gamut of kernel-level threats, but the list is not meant to be exhaustive due to space constraints. These rootkits in particular subvert function pointers and reachable data. Some rootkits burrow themselves very deep into the kernel and modify non-trivial data structures. To detect these rootkits, an object mapping can be created using a tool such as KOP[59] to identify malicious alterations.

DR:

The DR rootkit uses processor-level hardware debug facilities to intercept system calls rather than modifying the actual system call table itself. DR reprograms a hardware breakpoint that is reached every time a system call is made[60]. The breakpoint then intercepts the call and runs its own handler before passing control to the legitimate system call handler. Since Forenscope does not restore the state of debug registers, DR is effectively neutralized across the reboot, and as a result, hidden processes are revealed. Informant detects DR in several ways: DR is present in the module list, DR symbols are exported to the kernel and DR debug strings are present in memory. If an attacker modifies DR to make it stealthier by removing these indicators, we contend that it is still hard to deceive Forenscope, since the debug registers are cleared as part of the reboot process. Although Forenscope doesn't restore the contents of the debug registers faithfully, this doesn't pose a problem for most normal applications because generally only debuggers use this functionality.

Phalanx B6:

Table 7.4: Effectiveness against rootkit threats

Rootkit	Description	Sanitization action
DR	Uses debug registers to hook system calls	Rebooting clears debug registers
Phalanx b6	Uses /dev/kmem to hook syscalls	Restore clean syscall table
Mood-NT	Multi-module RK using /dev/kmem/	Clear debug regs, restore pointers
Adore	Kernel module hooks /proc VFS layer	Restore original VFS pointers

Phalanx hijacks the system call table by directly writing to memory via the */dev/mem* memory device instead of using a kernel module. It works by scanning the internal symbol table of the kernel and redirecting control flow to its own internal functions. Informant detects Phalanx while checking the system call table and common kernel pointers. Neutralizer restores the correct pointers to inoculate the system of Phalanx.

Adore:

Adore [61] is a classic rootkit which hijacks kernel pointers to deceive tools such as *ps* and *netstat*. It works by overwriting pointers in the */proc* filesystem to redirect control flow to its own functions rather than modifying the syscall table directly. Informant detects that the pointers used by Adore do not belong to the original read-only program code segment of the kernel and Neutralizer restores the correct pointers. Restoration of the original pointers is simple and safe because the overwritten VFS function operations tables point to static functions such as *proc_readdir*, while Adore has custom handlers located in untrusted writable kernel module address space.

Mood-NT:

Mood-NT is a versatile multi-mode rootkit that can hook the system call table, use debug registers and modify kernel pointers. Because of its versatility, the attacker can customize it for different purposes. Like the rootkits described previously, Forenscope detects Mood-NT in various modes. Our experiments indicate that Mood-NT hooks 44 system calls and Forenscope detects all of these alterations. Furthermore, each hook points out of the kernel's read-only program code address space and into the untrusted memory area occupied by the rootkit.

7.7 Size

Forenscope is written in a mixture of C and X86 assembly code. Table 7.5 shows that Forenscope is a very small program. It consumes less than 48 KB in code and 125 KB in running memory footprint. The lines of code reported in the table are from the output of the *sloccount* program. We break down the size of each component into core C and assembly code, hardware-specific restoration code and module code. To minimize its size, Forenscope reuses existing kernel code to reinitialize the disk and network; the size of this kernel code is device-specific and therefore excluded from the table, since these components are not part of Forenscope. The small compiled size of Forenscope and its modules implies that a minimal amount of host memory is overwritten when Forenscope is loaded onto the system. Furthermore, the diminutive size of the code base makes it more suitable for auditing and verification.

Table 7.5: Sizes of Forenscope and modules

Component	Lines of Code	Compiled Size (Bytes)
Forenscope (C)	1690	15,420
Forenscope (Assembly)	171	327
Forenscope (Hardware)	280	1,441
Neutralizer & ForenShell	34	8,573
Other Modules	861	22,457
Total	3,036	48,218

Chapter 8

Discussion

Live and volatile forensics techniques are still nascent in many ways. To make forward progress, there needs to be advancements made on four fronts. First, the legal community should come up with more precise definitions and standards regarding the collection of volatile evidence and its admissibility in court. Second, better capture techniques need to be developed to collect a higher fidelity snapshot of the original evidence. Third, evidence collection tools need to improve by being less intrusive and more complete. Finally, with these developments in place, analysis tools need to better identify interesting evidence and help the investigator reconstruct the "forensic timeline." This discussion section will touch upon some of the issues that need to be solved to realize these goals.

8.1 Legal Standards

Legally, the jury is still out on the use of live forensic tools because of the issues of taint and blurriness. While some recent cases [12] suggest that courts are starting to recognize the value of the contents of volatile memory, the validity of the evidence is still being contested. A recent manual on collecting evidence in criminal investigations released by the Department of Justice [62], instructs that *no limitations should be placed on the forensic techniques that may be used to search* and also states that use of forensic software, no matter how sophisticated, does not affect constitutional requirements. Although we do not make strict claims of legal validity in the courts, we are encouraged by the above guidelines to collect as much volatile information as possible. We objectively compare our tool against the state of the art and find that it does collect more forms of evidence with better fidelity than existing tools.

8.2 Acquisition Methods and Non-intrusive Capture

Current proposals to acquire forensic evidence fall into two broad categories: hardware and software-based tools. Generally speaking, in most cases including criminal investigation and incident response, hardware tools must be installed a priori. PCI acquisition cards such as CoPilot [63] need to be present on the system at the time of acquisition. Other forms of acquisition over Firewire [64] can be performed on a running machine if the proper prerequisites are met. These tools are generally limited by blurriness if they do not freeze a system under acquisition. Likewise, since they all access memory through a PCI bus, they are unable to inspect the contents of a CPU's cache. With today's large processor caches, this lack of visibility may cause issues with the freshness and completeness of any evidence collected.

In terms of software, the options are much less constrained. As discussed earlier, many of these tools run on the host operating system and thus depend on the TCB of the running system. This can pose problems when the lower layers are not necessarily trustworthy because of malware infections. Trustworthiness issues aside, these software solutions often induce taint by virtue of using operating system resources. Several variations of this technique have been attempted including booting acquisition kernels [65] and combing hibernation files [66]. While these variants allow for consistent high-fidelity snapshots to be taken, they do have several limitations. First, the use of an acquisition kernel or system crash dump is a one-way process. There is no way to recover the system from the induced crash, so the system must be restarted after acquisition. This may limit further acquisition prospects after the initial system memory dump. Secondly, although combing the hibernation file may be a practical measure, the act of hibernating the machine notifies the running software that the machine is about to go to sleep. This sleep signal can be caught by malware, encryption/privacy tools, VPNs and even applications. Modern software has a great deal of leeway to clean up sensitive information well before the machine goes to sleep. On the other hand, the hibernate file itself may be encrypted, which may present significant challenges to such tools.

8.3 Identifying and Examining Evidence

The process of identifying and extracting evidence using volatile memory forensics techniques is still nascent in many ways. Current techniques developed to extract evidence of interest often rely on expert knowledge or some intuition about the structure of evidence. These approaches have explored extracting ASCII data and data of high entropy. We believe that these approaches can be complemented by the use of statistical data to further identify these structures. Cafegrind is a step in this direction because it helps to assess the practicality of data extraction and automatically identify target types.

In the long term, we believe that the corpus of programs that need to be considered in a forensic investigation can be quite large, and better systematic approaches to forensics are necessary to address the natural diversity in software systems. In the process of doing so, it is absolutely necessary to establish a ground truth as a baseline to measure evidence extraction achievement against. Since many popular applications are written in loosely-typed languages, it becomes necessary to adopt type-inferencing and type-discovery methods to capture an accurate object map effectively.

Another trend that affects analysis is the use of modular components in software. For instance, many applications embed web browsers, movie players and encryption algorithms. Learning to recognize forensic evidence in one instance of a library is sufficient, because the same techniques can be equivalently applied to all other similar instances. However, this level of sharing illustrates how complicated the software stack can be. In the course of our analysis, we found that some applications such as web browsers link to many shared libraries, and in some cases such as Konqueror, the original application binary simply launches an instance of a shared web rendering framework called WebKit.

Furthermore, since libraries have strict function export interfaces and well-defined data structures for interaction, crossing library boundaries can reveal a wealth of forensic information. This is not surprising because these interfaces have been used as type-revealing operations in type-sink systems[67]. Likewise, the use of these libraries often requires data type conversion between different kinds of data structures and this results in duplication of data. This duplication happens at the data structure level where the structures may be shadowed in different libraries or in buffers where libraries pass information to each other. For instance, a web browser might use *libxml* to parse an XML file, and then use *libqt* to display it. An XML file parsed in this workflow would appear in the private data of both libraries. Additionally, data can be buffered when it is being compressed or encrypted. The original buffer contains a copy of the data as well as the compression/encryption library's temporary buffer. As a result, there are many copies of similar data present in memory and our analysis is just the first step in shedding some light on the associations between these structures.

Attribution is another important issue in this discussion. Some data structures found in web browsers may not have been directly created as a direct consequence of user actions. For instance, an advertisement served on a page may not have been knowingly requested by the user and any forensic analysis should take the source of such evidence into account. This effect is especially evident with Tor where anonymized network packets may be routed through client nodes. Any forensic evidence collected from Tor should take into account the source of the data with proper attribution. Some properties necessary for attribution include proper factorization of functionality into units where each unit belongs to a principal and explicit control and data flow to track activities. Further research needs to be done to ensure proper attribution of evidence.

In addition to attribution, logging and in-flight evidence collection are necessary to acquire enough data points to create a forensic timeline. Approaches such as [68, 69] provide trustworthy logging to aid the collection and preservation process. Once enough points are collected and attributed, tools such as [70, 71] can be used to help generate the forensic timeline automatically. Improved facilities to collect, attribute and correlate these events would be highly desirable in next-generation operating systems and distributed systems to support forensic activities.

8.4 Trustworthiness

While evaluating Forenscope, we observed different behavior of rootkits on virtual machines and physical hardware. Our observations confirm the results of Garfinkel et al [72] that virtual machines cannot always emulate intricate hardware nuances faithfully and as a result some malware fails to activate on a virtual machine. For example, malware such as the Storm worm and Conficker[73] intentionally avoid activation when they sense the presence of virtualization to thwart the analysis process. Hence analyzing a system for rootkits using a virtual machine may not only cause some rootkits to slip under the radar but also alert them to detection attempts. Since Forenscope continues to run the system without exposing any of the issues raised by running virtualization systems, we argue that the system is unlikely to tip off an attacker to the presence of forensic software. More recently, some tools such as MAVMM[74] have been developed for the express purpose of analyzing malware, but they must be installed a priori and they also require the presence of virtualization hardware.

Although it is possible for attackers to falsify or fabricate information to impede the examination process, a good forensic investigator will usually be able to detect an inconsistency in the data. Locard's principle of exchange suggests that it is very difficult for an attacker to remove all traces of a crime. For instance, an attacker may alter system log files in an attempt to hide their tracks, but they may forget to clear the SSH log file, shell history, open network sockets, or latent volatile data structures in the kernel may suggest that the system has been compromised. Detection of these inconsistencies using manual approaches has been effective in practice and automating these practices using tools such as [70, 71] can be promising.

Inconsistencies detected in various data sources can act as indications that certain artifacts collected may have a low level of trustworthiness. For instance, if an attacker has gained user-level access to a machine, then files found in the user's home folder may have been altered. Alternatively, if a suspicious kernel module is found, little if any information found on the machine may be trustworthy. Table 8.1 outlines various scenarios relating to trust.

Alternatively, attackers may decide to target the forensic tools themselves to crash or take control of them to falsify or hide information. This is not surprising since software inherently has bugs and security vulnerabilities. Complicated code that parses data and processes intricate data structures can be vulnerable to format string vulnerabilities, buffer overflows and logic errors. Since Forenscope is relatively small at less than 5,000 lines of code, it may be possible to formally verify the code base. To make the code base small, Forenscope reuses large amounts of code from the bootloader, Grub, and also makes extensive use of BIOS calls. Formal analysis would also have to consider these dependencies to ensure correctness and trustworthiness.

8.5 Countermeasures

Although Forenscope provides in-depth forensic analysis of a system in a wide variety of scenarios, there are countermeasures that attackers and criminals can use to counter the

Scenario	Description
Compromised user account	Files owned by the user may not be trustworthy
Root access acquired	The attacker may have altered log files or reconfigured the system. Logs and files may not be trustworthy.
Kernel level rootkit found	Kernel level data structures are not trustworthy
Log files inconsistent	The attacker may have altered a system logfile to cover their tracks, but forgot to clean another (debug) logfile
Log files inconsistent with shell history	The attacker may have altered a system logfile to cover their tracks, but forgot to clear the shell history
/proc inconsistent with ps or netstat	The attacker may have replaced system admin utilities with trojan versions
Kernel data structures disagree with ps or netstat	Attacker may have altered kernel data structures. Use cross-view diff such as Strider-Ghostbuster to reveal rootkit.

Table 8.1: Trustworthiness of evidence

use of Forenscope. From an incident response perspective, we assume that the machine is controlled by the owner and that the attacker does not have physical access to it. This means that only software-based anti-forensic techniques are feasible, although some of these techniques may involve changing hardware settings through software. Most of the hardware and software state involved in these anti-forensic techniques are cleared upon reboot or rendered harmless in Forenscope's clean environment. In investigation, the adversary may elect to use a BIOS password, employ a secure boot loader, disable booting from external devices or change BIOS settings to clear memory at boot time. These mitigation techniques may work, but if the investigator is sophisticated enough, he can try techniques suggested by Halderman et al [35] to cool the memory chips and relocate them to another machine that is configured to preserve the contents of DRAM at boot time. One other avenue for working around a password-protected BIOS is to engage the boot loader itself. We found that some boot loaders such as GRUB allow booting to external devices even if the functionality is disabled in the BIOS. The only mitigation against this channel is to use password protection on GRUB itself, which we believe is not frequently used.

Alternately, some machines purposefully disable the USB ports in an attempt to prevent the introduction of foreign software. This practice impedes the use of forensics software. However, these policies are usually enforced at the OS level where Windows has been configured to disallow external USB devices, even though the hardware and BIOS have been configured to allow USB devices. In this case, Forenscope can circumvent the Windows-based security policy because Windows is not running to enforce these access controls.

8.6 Limitations

The only safe harbor for malware to evade Forenscope is in conventional memory itself because the act of rebooting pollutes the contents of the lower 640 KB of memory considerably thus potentially erasing evidence. However, we contend that although this technique is possible, it is highly unlikely for three reasons: first, for such malware to persist and alter the control flow, the kernel must map in this memory area in the virtual address space. This requires a change in the system page tables that is easily detectable by Forenscope since most modern operating systems do not map the conventional memory space into their virtual memory space. Secondly, such malware would have to inject a payload into conventional memory and if the payload is corrupted by the reboot process, the system will crash. Finally, such malware won't survive computer hibernation because conventional memory is not saved in the process. Even if Forenscope is unable to restore the system due to extenuating circumstances, we still have an intact memory dump and disk image to analyze.

In terms of fragility, Forenscope is versatile enough to adjust to the starting address of global variables in the kernel as it looks them up using `System.map` in Linux and leverages symbol lookup facilities. However, like existing examination tools such as Volatility, Forenscope is sensitive to the internal layout and offset of kernel data structures. Most core data structures retain their layout in minor point release software revisions, however, major releases can break compatibility. When this happens, internal circuit breakers (assertions) within Forenscope halt execution if a major inconsistency is found. Handling major changes in data structure layout is a hard problem faced by the debugging and reverse engineering communities. Generally, major tools cannot adjust to these changes automatically, but recent advances with adaptive reverse engineering tools/debuggers such as IDA Pro offer promising results. Recent advancements [75, 76] can semi-automatically infer data structure offsets by using instrumentation and disassembly of target code. This appears to be a promising approach to determine necessary data structure offsets to enable the use of tools such as Forenscope.

Recent advances in storage technology and privacy guards may necessitate a greater research investment in volatile forensic tools. For instance, the adoption of SSD drives threatens the use of traditional forensics tools that operate on magnetic media because these drives use firmware that eagerly clears deleted data [77]. NAND-based drives must first erase a disk block before the block can be written to. In order to optimize the process, drive manufacturers have produced firmware that clears unused disk blocks in the background. This process naturally erases residual forensic evidence. On the contrary, NAND drives may also offer new hope for forensics as bad blocks are remapped to different areas and existing privacy tools do not adequately handle these cases because they can fail to clean up old replicas[78]. Phase Change Memory (PCM) is likely to encounter similar issues [79, 80].

Chapter 9

Related Work

The Forenscope framework uses many technologies to achieve a high-fidelity forensic analysis environment through introspection, data structure analysis and integrity checking. Many of the introspective techniques used by Forenscope were inspired by similar functionality in debuggers and simulators. VMware's VMsafe[81] protects guest virtual machines from malware by using introspection. A virtual machine infrastructure running VMsafe has a security monitor that periodically checks key structures in the guest operating system for alteration or corruption. Projects such as Xenaccess[6] take the idea further and provide a way to list running processes, open files and other items of interest from a running virtual machine in a Xen environment. Although Xenaccess and Forenscope provide similar features, Xenaccess depends on the Xen virtual machine monitor, but the investigator cannot rely on its presence or integrity. On some older critical infrastructure machines, legacy software requirements make it impractical to change the software configuration. Forenscope does not have such requirements. Forenscope's techniques to recover operating system state from structures such as the process list have been explored in the context of analyzing memory dumps using data structure organization derived from reverse-engineered sources[82, 83]. Cozzie et al[84] used unsupervised learning to detect data structures in memory while [85, 86, 87] reverse engineer data formats using automated extraction and mining techniques. Attestation shows that a machine is running with an approved software and hardware configuration by performing an integrity check. Forenscope builds upon work pioneered by the virtual machine introspection community to allow forensic analysis of machines that are not prepared a-priori for such introspection. It provides a transparent analysis platform that does not alter the host environment and Forenscope supports services such as BitBlocker that allow an investigator to explore a machine without worry of inducing taint.

The techniques used by Forenscope for recovering running systems are well grounded in the systems community and have been studied previously in different scenarios. The original Intel 286 design allowed entry into `protected mode` from `real mode`, but omitted a mechanism to switch back. Microsoft and IBM used an elegant hack involving memory remanence to force re-entry into real mode by causing a reboot to service BIOS calls. This technique was described by Bill Gates as "turning the car off and on again at 60 mph" [88]. Some telecommunications operating systems such as Chorus [89] are designed for quick recovery after a watchdog reset and simply recover existing data from the running operating system rather than starting afresh. David [40] showed that it is possible to recover from resets triggered by the watchdog timer on cell phones. BootJacker [36] showed that it is possible for attackers to recover and compromise a running operating system by using a carefully crafted forced reboot. Forenscope applies these techniques in the context of forensic analysis and our work presents the merits and limitations of using such techniques to build a forensic tool.

An alternative to state recovery techniques is to corrupt the state saved by the operating system as part of the hibernation process. Sandman[90] takes this approach by intentionally patching the hibernation file of a sleeping machine. The authors reverse engineered the compressed hibernation file format used by Microsoft Windows and patched the OS to circumvent security measures. While this approach may be less brittle than state-restoration techniques because it uses a supported operating system vector, it does have several limitations. First, when an operating system begin to hibernate, running applications are notified of the hibernation request, which gives them an opportunity to clean up any secrets they may have in memory. Second, networked applications tend to disconnect current sessions. We have observed this behavior for SSH, VPN and remote desktop sessions. In contrast, Forenscope's forced reboot skips over this notification process. Finally, the act of hibernation creates entries in the system log and activates a system code path that wouldn't be invoked in the normal case, causing taint and other undesired side effects along the way.

Devices such as the Trusted Platform Module and Intel trusted execution technology (TXT) provide boot time and run-time attestation respectively. Although TPM may be available for some machines, the protection afforded by a TPM may not be adequate for machines that are meant to run continuously for months. These machines perform an integrity check when they boot up, but their lengthy uptime results in a long time of check to time of use (TOCTTOU) that extends the duration for breaches to remain undetected. Hardware solutions such as Copilot[63] are available to check system integrity. In contrast, Forenscope performs an integrity assessment at the time of use; which allows the investigator to collect evidence with better fidelity.

Chapter 10

Concluding Remarks and Future Work

Forenscope explores live forensic techniques in the context of evidence preservation, non-intrusiveness and fidelity that concern such approaches. Measured against existing tools, our experiments show that Forenscope can achieve better compliance within the academic guidelines prescribed by the community. Forenscope shows that volatile state can be preserved and that the techniques embodied in Forenscope are broadly applicable to modern operating systems such as Windows and Linux. We encourage further development of tools based on our high-fidelity analysis framework and believe that it can enable the advancement of the state of the art.

Thorough evaluation of our techniques has shown that they are safe, practical and effective by minimally tainting the system, while causing negligible disruption to critical systems. We believe that these techniques can be used in cases where traditional tools are unable to meet the needs of modern investigations. Furthermore, we have shown that our techniques provide the benefits of live forensics with the trustworthiness of dead box forensics. To continue the development of this tool, we plan to work closely with partners to better evaluate use of this tool in real-world scenarios such as incident response in a variety of contexts.

Appendix A

Cafegrind

Cafegrind[1] is a memory analysis tool designed to help programmers and forensic analysts better understand how memory is allocated and used within a program. As an extension to Valgrind, it naturally supports all of the inherent memory checking abilities of the host tool. The mission of Cafegrind differs from that of Valgrind in the sense that Cafegrind is meant to understand the structure of data types and their associated access patterns. In this dissertation, we use Cafegrind to measure the longevity of data structures in memory after they have been relinquished by the free() or delete call. We also explore access patterns to better quantify the velocity at which data structures change. These metrics provide a measure of fidelity for evidence. Structures that change infrequently are likely to have lower blurriness and higher fidelity of capture. On the other hand, structures that are frequently updated are likely to be subject to the effects of blurriness. We believe that the ability to quantify these effects is essential to making an argument about the admissibility of fleeting volatile evidence.

Unlike TaintBochs[31], we utilize a more semantically aware tracking approach that respects actual data types. TaintBochs performs tracking at the byte level and is thus data structure-agnostic. Similarly, Laika [84] attempts to infer data types using Bayesian analysis. Our approach uses intrinsic debugging information to track data types rather than treating the machine as a blackbox. The limitations of our approach include: tracking only user-level memory areas and the need for the target program to be compiled in debug mode.

[1] Cafegrind: **C**/C++ **A**nalysis **F**orensic **E**ngine for (Val)grind

A.0.1 Cafegrind design

Valgrind

Cafegrind is designed as an extension to Valgrind [91], a suite of tools for debugging and profiling. Common functionalities of Valgrind include memory leak detection, cache simulation and program analysis to detect concurrency bugs. Valgrind executes target programs by dynamically translating them into its internal VEX execution format. As a result, Valgrind is able to perform fine-grained instrumentation at the lowest levels of the machine architecture. Unlike similar emulation systems such as QEMU, Valgrind is also able to interpret higher-level debugging symbol information to support various functionalities such as memory leak detection. Cafegrind builds upon these intrinsic features to track the lifecycle of data and provides additional insight into specific data structures by performing automatic type inferencing.

The Lifecycle of Data

To better understand how evidence can be found in memory, we must first understand the lifecycle of data. The lifecycle of data in a program is shown in Figure A.1. First, data is allocated by a function such as malloc() or `new`, then it is initialized by a function such as memset(), C++ constructor or memory pool constructor. Once the base object is ready, its fields are populated with information and the structure is accessed and modified as the program runs. Once the structure is no longer needed, it is freed and the memory returns to a pool for reallocation. Throughout this process, evidence can be erased by modification, initialization and reallocation. However, the process of relinquishing memory does not always clear the latent contents of the data structure. In many cases, data is only partially destroyed, as reuse of a memory area does not always completely overwrite old data. This partial destruction process is the underlying principle behind volatile memory forensic analysis and is useful in uncovering freed data. Cafegrind uses empirical methods to track

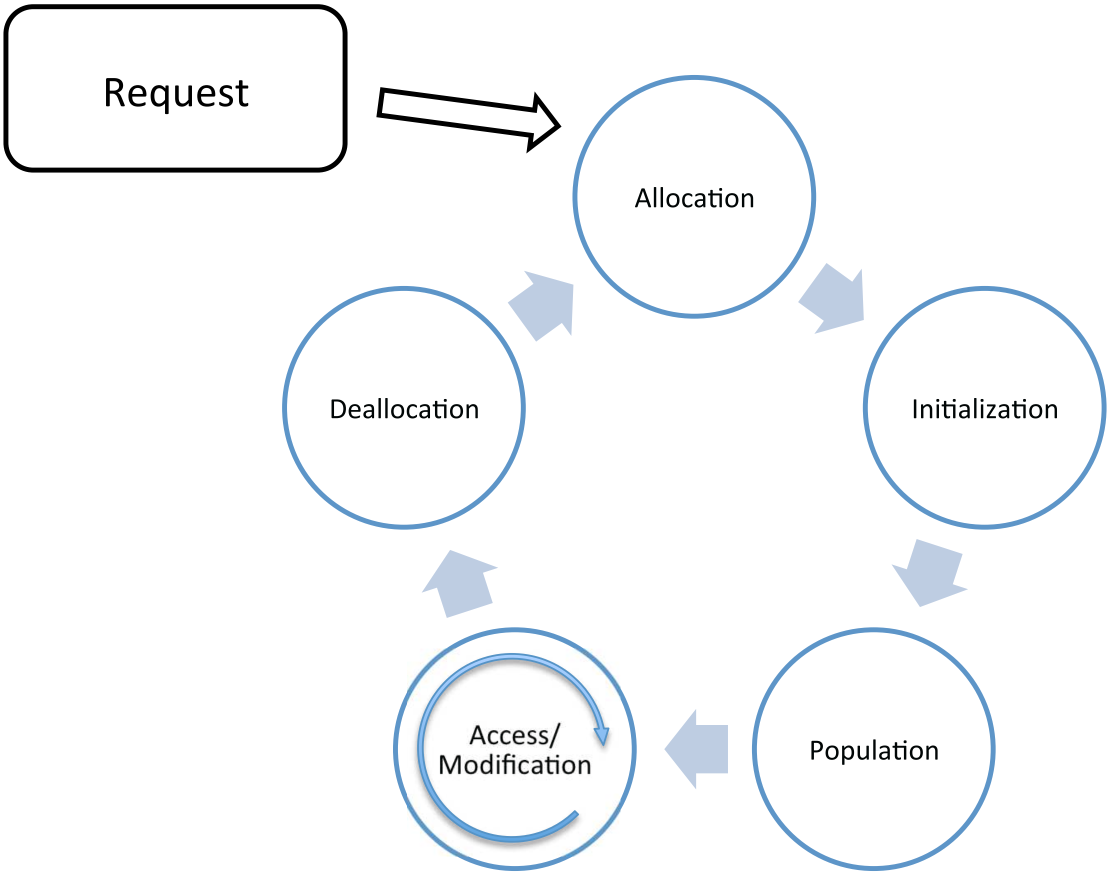

Figure A.1: The lifecycle of data

how much data can be recovered from memory dumps that contain active and freed data.

Type Inferencing

Since C/C++ are not strongly typed languages, Cafegrind must infer the type of allocated memory areas to build its object map. To illustrate how this works, consider the following code snippet:

```
[1] struct datastructure *mydata;
[2] mydata = (struct datastructure *)
malloc( sizeof( struct datastructure ) );
[3] mydata.fld1 = 100;
```

In the above example, on line 1, the program allocates space on the stack to store a pointer to an object of type `datastructure`. Line 2 calls malloc() to allocate memory for type `datastructure`, but malloc() returns a memory area with the type void* which is then cast back to the proper type. This type casting is not present in the assembly code, so we have to rely on an alternate method to infer the type. The key assignment here is on line 2. When Cafegrind observes that the malloc memory area is assigned to a local stack variable, Cafegrind propagates the type information from the stack variable to the memory area. In assembly code, line 2 is implemented by using a store instruction where the target of the store is the address of the stack variable `mydata` and the value written to it is the address returned by malloc(). Cafegrind first checks if the type for the target address is known and having determined that the type is `struct datastructure`, we can propagate the type to the memory area described by the value in the instruction. Cafegrind uses this incremental process to build and maintain an object map in memory.

Alternately, in C++, types can be inferred by monitoring object constructors. When *new* is invoked, all constructors in an object's inheritance hierarchy are called starting from the most generic to the most specific. For instance, a *square* object may run its constructors in the following order: shape→rectangle→square. In this case, the object type inference would be made when the square constructor is called with a pointer to a square implicitly passed with the *this* argument. Since C++ stores its constructors in a special ELF[92] section called **.ctors**, we can adequately identify which functions serve as constructors for type inferencing. Another safety check is to ensure that the name of the constructor is consistent with the purported type of the object.

Methodology

In order to track the data lifecycle of a program and build an object map by performing type inferences, Cafegrind instruments the following events while a program is running:

1. Memory allocation
2. Data structure read/write accesses
3. Memory deallocation

In order to support type inferencing, Cafegrind intercepts all memory allocation and deallocation requests. When an allocation is made, Cafegrind tracks the memory object returned by malloc() and the stack backtrace at the time of the allocation. Recall that malloc() does not return typed objects; objects are of the generic type void*. These allocations are stored in an efficient ordered set data structure for fast lookup and retrieval. Cafegrind only discovers the type of a memory object when the object is loaded into a typed pointer object as described in Section A.0.1. This is enabled by intercepting store instructions to memory locations and Cafegrind can thus track the assembly level assignment of a pointer to the memory object to a stack location. If the pointer points to a tracked memory location, Cafegrind performs a lookup on the debug information associated with the executable to describe the type of the stack variable. This process queries the debug information present in loaded libraries and binaries in the DWARF3 [93] format and helps identify the type of the stack object. Once the type of the stack object has been identified, it is then propagated to the dynamically allocated memory object and stored in the same ordered set.

In addition to performing type inferencing, Cafegrind also tracks accesses and modifications to the data structure. This allows analysis of the access patterns to a particular type. These accesses are tracked by instrumenting all memory loads and stores. In the previous code snippet, line 3 illustrates how Cafegrind tracks data accesses. When a memory address is accessed, Cafegrind checks its internal allocation database to see whether or not the access belongs to a tracked allocation. If so, Cafegrind identifies which allocation the access belongs to and resolves the member being accessed. These accesses are tracked and aggregated statistically to reveal how the underlying data types are being used. Further, we also track which function call led to the data access and aggregate this information to find which data types a particular function accesses. Cafegrind once again uses efficient ordered sets to perform the lookup and updates quickly.

Cafegrind also maintains a set for allocations that have been freed by the application. This set is used to track when objects are overwritten in memory and helps determine the time at which data is destroyed. In addition to tracking the properties of data structures, we also collect their contents. This is useful for offline analysis using utilities like `strings` and the contents are clustered by their type in separate files to enable easier processing.

By performing such monitoring, Cafegrind is able to better understand how data is created, accessed and destroyed. This can provide an empirical analysis on which data structures can be expected to persist and for how long. For instance, a forensic analyst may find some interesting information in an HTML cache object, but this type of object may be transient and the data stored in it can change throughout a browsing session as the user visits certain websites. Cafegrind can provide an analysis of the longevity of such data.

A.0.2 Observations

Experimental Setup

Our technique described in Section A.0.1 relies on explicitly monitoring canonical variable accesses and assignments. Common compiler optimizations can store pointer values in registers instead of allocating space on the stack. This process can disturb Cafegrind's type inferencing algorithm, so Cafegrind only works on binaries and shared libraries that are compiled with debugging enabled and optimization disabled. From a forensic standpoint, an investigator is unlikely to encounter a machine with such a configuration. However, the purpose of this paper is to study the ideal behavior of data structures, and applying these alterations does not operationally affect the behavior of the applications we study except for imposing additional runtime overhead when the program is being instrumented.

We used two test machines in our Cafegrind experiments. Our first test machine has two quad core Intel Xeon E5410 CPUs running at 2.33 Ghz with 4 GB of RAM. Our second test machine has a single Intel Core i7 920 CPU running at 2.67 GHz with 6 GB of RAM. Both machines are running Gentoo Linux 1.12.14 on the 2.6.34-r12 kernel. All the applications and libraries are compiled with debugging enabled and optimization disabled. This configuration affords Cafegrind visibility into the inner workings of system applications and libraries. As a result, we can trace the entire execution chain of entire subsystems such as the KDE desktop environment if desired.

Basic Concepts

For each object, we track the following attributes:

1. Type - The type of an object
2. Size - The size of an object
3. Age - The length of time an allocation lasts before it is deallocated
4. FreedAge - The length of time a deallocated structure lasts before it is clobbered by a subsequent allocation
5. Reads - Number of reads performed
6. Writes - Number of writes performed
7. Size - Size of the allocation

Our evaluation methodology cross-analyzes these attributes to find correlations which reveal patterns and relationships in the way that these structures are allocated, accessed and freed.

Table A.1: Coverage

Application	Store Coverage	Load Coverage	Overall Coverage
Firefox	70.48 %	88.11 %	83.51 %
KWrite	85.8 %	94.27 %	92.66 %
Links	99.09 %	99.99 %	99.6 %
Tor	85.95 %	96.43 %	95.02%

We measured forensically interesting data structures in several ways. First, we apply well-known string identification algorithms to find ASCII strings in memory. This helps us identify web pages and XML documents as well as HTTP requests that are latent in memory. Secondly, we perform basic outlier analysis on the dataset to highlight objects that are large, frequently accessed, or have great longevity. Other attributes are also considered in this process. Cafegrind then produces a list of candidate types that the forensic analyst can inspect more closely to find evidence of interest.

The precise list of identifiers is:

1. ASCII content - Structures with a large (configurable) percentage of ASCII characters
2. Entropy - Incompressible structures such as images, audio and encryption keys
3. Pointer content - Important structures are likely to be linked to other important structures in the object graph

In this paper, we have taken the candidate list and applied our expertise to identify and measure the characteristics of interesting types of evidence.

Correctness

The type inferencing coverage is measured by counting the number of load/store operations on dynamically allocated regions of memory for which Cafegrind has inferred the type. In terms of type inferencing coverage, Cafegrind performs remarkably well. As shown in Table A.1, we have seen upwards of 90% type inferencing coverage for many real world applications. This behavior should be expected since the type information comes from debug information ingrained in the program itself. The coverage was also improved as the system used for evaluation contained shared libraries built with debug information. Additionally since we tracked load/store instructions from the program and its linked libraries, our type inferencing technique was able to get a clear insight into the creation and usage of data types. There are several factors that can adversely affect the performance of the type inferencing system:

1. Generic arrays of char* and similar types that are cast into specific types before access
2. Variable length structures where the last element is a void* or char*
3. Unions used in structures where the type is ambiguous
4. Nested types with unions. Cafegrind currently doesn't handle nested type agreement.
5. The use of custom allocators which return void*. Cafegrind isn't able to follow multiple levels of type propagation currently

Nonetheless, Cafegrind still performs very well in real-world scenarios. Certainly, there are cases where programs use certain practices that may be challenging for type inferencing systems to handle. As we've seen with Firefox and QT, there are large codebases that use pointer wrappers, templating and other constructs which can obscure the true type of an object. This remains a challenge for even the best type inferencing systems. In the absence of compiler assistance, there aren't many options to handle these cases. For example, KOP [59] relies on additional compiler information to extract the type assignments. We do acknowledge that Cafegrind was not extensively tested against a ground truth, primarily because of the difficulty of obtaining a valid ground truth. However, from experience and cross validation, we believe that we've obtained fairly useful results.

Table A.2: Performance

Application	Time
Firefox native	6 s
Firefox Cafegrind	3:30 m
Firefox Lackey	2:57 m
Firefox Memcheck	1:52 m
Konqueror	2 s
Konqueror Lackey	3:12 s
Konqueror Cafegrind	3:26 s

A.0.3 Code Metrics

Cafegrind is over 2,100 lines of code and it builds upon the Valgrind framework, which has over 87,000 lines of code. Adding tracing and instrumentation functionality in Cafegrind is relatively straightforward by using standard Valgrind functions.

A.0.4 Performance

We compared the performance of Cafegrind against the performance of the Valgrind tool itself in Table A.2. Valgrind in its purest form imposes a modest performance penalty because it does binary translation and intercepts function calls. When Valgrind's Lackey full binary instrumentation mode is activated, the performance quickly degrades as each instruction is individually executed with memory tracing support enabled. Since Cafegrind uses facilities that are similar to what Lackey does, its performance is similar to that of Lackey. The additional functionalities of type inferencing and object tracking consume around 29% and 7% of execution time respectively for Tor and Konqueror. For Firefox, inferencing and tracking consume 14% and 11% respectively.

References

[1] A. Savoldi and P. Gubian. Blurriness in Live Forensics: An Introduction. In *Advances in Information Security and Its Application: Third International Conference, ISA 2009, Seoul, Korea, June 25-27, 2009. Proceedings*, page 119. Springer, 2009.

[2] Electronic Crime Scene Investigation: A Guide for First Responders. `http://www.ncjrs.gov/pdffiles1/nij/219941.pdf`, 2008.

[3] SANS Top 7 New IR/Forensic Trends In 2008. `http://computer-forensics.sans.org/community/top7_forensic_trends.php`.

[4] *Prosecuting Computer Crimes*, pages 141–142. US Department of Justice, 2007.

[5] Kristine Amari. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory, 2009.

[6] B.D. Payne, M.D.P. de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, 2007.

[7] International Organization on Computer Evidence. `http://www.ioce.org`, 2002.

[8] Scientific Working Group on Digital Evidence. `http://www.swgde.org`, 2005.

[9] Good Practice Guide for Computer Based Electronic Evidence, 2005.

[10] Eoghan Casey. *Digital Evidence and Computer Crime*. Academic Press, 2005.

[11] Keith Inman and Norah Rudin. *Principles and Practice of Criminalistics: The Profession of Forensic Science*. CRC Press, 2001.

[12] Columbia Pictures Indus. v. Bunnell, U.S. Dist. LEXIS 46364. C.D. Cal. `http://www.eff.org/cases/columbia-pictures-industries-v-bunnell`, 2007.

[13] United States National Institute of Justice Technical Working Group for Electronic Crime Scene Investigation. *Electronic Crime Scene Investigation: A Guide for First Responders*. 2001.

[14] Fred Smith and Rebecca Bace. *A Guide to Forensic Testimony*. Addison Wesley, 2003.

[15] Gary Palmer. *A Roadmap for Digital Forensic Research*. Technical Report. DFRWS., 2001.

[16] Brian D. Carrier. *A hypothesis-based approach to digital forensic investigations*. PhD thesis, West Lafayette, IN, USA, 2006.

[17] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley, 1997.

[18] D. Brezinski and T. Killalea. Guidelines for Evidence Collection and Archiving. RFC 3227 (Best Current Practice), February 2002.

[19] Guidance Software. EnCase Enterprise. `http://www.encase.com/`.

[20] Microsoft. Microsoft COFEE. `https://cofee.nw3c.org/`.

[21] e-fense. Helix3. `https://www.e-fense.com`.

[22] AccessData. FTK Imager. `http://www.accessdata.com`.

[23] A. Walters, N.L. Petroni Jr, and I. Komoku. Volatools: integrating volatile memory forensics into the digital investigation process. *Black Hat DC*, 2007, 2007.

[24] Mandiant. Memoryze. `http://www.mandiant.com/software/memoryze.htm`.

[25] G.G. Richard III and V. Roussev. Scalpel: a frugal, high performance file carver. In *Proceedings of the 2005 digital forensics research workshop (DFRWS 2005)*, 2005.

[26] Ellick Chan, Shivaram Ventakaraman, Nadia Tkach, Kevin Larson, Alejandro Gutierrez, and Roy Campbell. Characterizing Data Structures for Volatile Forensics. In *Systematic Approaches to Digital Forensic Engineering*. IEEE, 2011.

[27] Gecko. `https://wiki.mozilla.org`, 2011.

[28] S. Chen, H. Chen, and M. Caballero. Residue objects: a challenge to web browser security. In *Proceedings of the 5th European conference on Computer systems*, pages 279–292. ACM, 2010.

[29] The WebKit Open Source Project. `http://http://www.webkit.org/`, 2011.

[30] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[31] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, page 22. USENIX Association, 2004.

[32] Mozilla Developer Network. XPCOM. `https://developer.mozilla.org/en/XPCOM`, 2011.

[33] Walter Link and Herbert May. Eigenshaften von MOS-Ein-Transistorspeicherzellen bei tieften Temperaturen. In *Archiv fur Elektronik und Ubertragungstechnik*, pages 33–229–235, June 1979.

[34] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–90, July 1996.

[35] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, and Joseph A. Calandrino. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[36] E.M. Chan, J.C. Carlyle, F.M. David, R. Farivar, and R.H. Campbell. BootJacker: Compromising Computers using Forced Restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 555–564. ACM New York, NY, USA, 2008.

[37] U. Drepper. What every programmer should know about memory. *http://people.redhat.com/drepper/cpumemory.pdf*, 2007.

[38] Open Source Community. Bochs IA-32 Emulator Project. `http://www.gnu.org/software/grub/`.

[39] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[40] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[41] Microsoft. Running Nonnative Applications in Windows 2000 Professional. Microsoft. `http://technet.microsoft.com/en-us/library/cc939094.aspx`.

[42] GNU. GRand Unified Bootloader. `http://www.gnu.org/software/grub/`.

[43] Muhammed Ali Mazidi and Janice G. Mazidi. *80x86 IBM PC and Compatible Computers: Assembly Language, Design, and Interfacing; Volume I and II*. Prentice Hall PTR, Upper Saddle River, NJ, USA, fourth edition, 2002.

[44] Ramdisks - Now we are talking Hyperspace! `http://www.linux-mag.com/cache/7388/1.html`, June 2009.

[45] Charles Cazabon. MemTester. `http://pyropus.ca/software/memtester/`.

[46] MPlayer. `http://www.mplayerhq.hu/`.

[47] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational), July 1999.

[48] Clemens Fruhwirth. New Methods in Hard Disk Encryption. Technical report, Vienna University of Technology, June 2005.

[49] David Bryson. The Linux CryptoAPI: A User's Perspective, May 2001.

[50] Juho Mäkinen. Automated OS X Macintosh password retrieval via firewire. `http://blog.juhonkoti.net/2008/02/29/automated-os-x-macintosh-password-retrieval-\via-firewire`, 2008.

[51] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). *ACM Transactions on Information Systems Security*, 7(2):319–332, 2004.

[52] John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. In *ICICS '97: Proceedings of the First International Conference on Information and Communication Security*, pages 233–246, London, UK, 1997. Springer-Verlag.

[53] John Loughran and Tom Dowling. A Java implemented key collision attack on the data encryption standard (DES). In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 155–157, New York, NY, USA, 2003. Computer Science Press, Inc.

[54] Stanislaw Jarecki, Nitesh Saxena, and Jeong Hyun Yi. An attack on the proactive RSA signature scheme in the URSA ad hoc network access control protocol. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 1–9, New York, NY, USA, 2004. ACM.

[55] Fuzen Op. The FU rootkit. `http://www.rootkit.com/project.php?id=12`.

[56] Sherri Sparks and Jamie Butler. Raising The Bar for Windows Rootkit Detection. *Phrack*, 11(63), 2005.

[57] D.A. Dai Zovi. Hardware Virtualization Rootkits. *BlackHat Briefings USA, August*, 2006.

[58] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[59] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565, New York, NY, USA, 2009. ACM.

[60] Edge, Jake. DR rootkit released under the GPL. `http://lwn.net/Articles/297775/`.

[61] The Adore Rootkit. `http://stealth.openwall.net/`.

[62] DOJ. *Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations*, pages 79,89. Computer Crime and Intellectual Property Section Criminal Division, 2009.

[63] N.L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh. Copilot-A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[64] Adam Boileau. Hit By A Bus: Physical Access Attacks with Firewire. In *RUXCON*, Sydney, Australia, Sep 2006.

[65] B. Schatz. BodySnatcher: Towards reliable volatile memory acquisition by software. *Digital Investigation*, 4:126–134, 2007.

[66] Matthew Suiche. Enter Sandman (why you should never go to sleep). In *PacSec07*, 2007.

[67] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS10), San Diego, CA*, 2010.

[68] M. LeMay and C. Gunter. Cumulative attestation kernels for embedded systems. *Computer Security–ESORICS 2009*, pages 655–670, 2010.

[69] C.R. Johnson, M. Montanari, and R.H. Campbell. Automatic Management of Logging Infrastructure. In *CAE Workshop on Insider Threat*.

[70] S.T. King and P.M. Chen. Backtracking intrusions. *ACM SIGOPS Operating Systems Review*, 37(5):223–236, 2003.

[71] A. Case, A. Cristina, L. Marziale, G.G. Richard, and V. Roussev. FACE: Automated digital evidence discovery and correlation. *Digital Investigation*, 5:65–75, 2008.

[72] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[73] B. Zdrnja. More tricks from Conficker and VM detection. `http://isc.sans.org/diary.html?storyid=5842`, 2009.

[74] A.M. Nguyen, N. Schear, H.D. Jung, A. Godiyal, S.T. King, and H.D. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *2009 Annual Computer Security Applications Conference*, pages 441–450. IEEE, 2009.

[75] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

[76] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection.

[77] G.B. Bell and R. Boddington. Solid State Drives: The Beginning of the End for Current Practice in Digital Forensic Recovery? *Journal of Digital Forensics, Security and Law*, 5(3):1–20, 2010.

[78] S. Swanson and M. Wei. Safe: Fast, verifiable sanitization for SSDs, 2010.

[79] S. Raoux, GW Burr, MJ Breitwisch, CT Rettner, Y.C. Chen, RM Shelby, M. Salinga, D. Krebs, S.H. Chen, H.L. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[80] RF Freitas and WW Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, 2008.

[81] VMware VMsafe Security Technology. `http://www.vmware.com/go/vmsafe`.

[82] Andreas Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. The Proceedings of the 6th Annual Digital Forensics Research Workshop, 2006.

[83] B. Dolan-Gavitt. The VAD tree: A Process-eye View of Physical Memory. *Digital Investigation*, 4:62–64, 2007.

[84] A. Cozzie, F. Stratton, H. Xue, and S.T. King. Digging for Data Structures. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[85] J. Okolica and G.L. Peterson. Windows operating systems agnostic memory analysis. *digital investigation*, 7:S48–S56, 2010.

[86] A. Case, L. Marziale, C. Neckar, and G.G. Richard III. Treasure and tragedy in kmem_cache mining for live forensics investigation. *Digital Investigation*, 7:S41–S47, 2010.

[87] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium, San Diego, CA*. Citeseer, 2008.

[88] Jerry Pournelle. OS — 2: What is is, What is isn't – and some of the Alternatives. *Infoworld*, 1988.

[89] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1991.

[90] N. Ruff and M. Suiche. Enter Sandman (why you should never go to sleep). In *PacSec applied security conference*, 2007.

[91] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, 2007.

[92] Executable and Linking Format. `http://refspecs.freestandards.org/elf`, 1995.

[93] DWARF 3.0 Standard. `http://dwarfstd.org/Dwarf3Std.php`, 2005.

Author's Biography

Ellick M. Chan was born in Hong Kong, China on February 10, 1981. He graduated with highest honors from the University of Illinois in 2004 with a joint masters and bachelor's degree in Computer Science. After graduation, he continued his doctoral studies at Illinois in Computer Science while also working at Motorola and earning a joint MBA degree along with the Ph.D. In 2011, he obtained a doctoral degree from the University of Illinois at Urbana-Champaign while working under the supervision of Professor Roy H. Campbell. His core research is on systems-level techniques to perform high-fidelity live forensics and he has presented the results of his research at several influential conferences. His work on hardware supported malware received the best student paper award at the prestigious IEEE Symposium on Security and Privacy held at Oakland in 2008.