# Forensic Acquisition and Analysis of Volatile Data in Memory

## Forensische Sicherung und Auswertung flüchtiger Daten im Hauptspeicher

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Stefan Vömel aus Limburg an der Lahn

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

| | |
|---|---|
| Tag der mündlichen Prüfung: | 18.12.2013 |
| Vorsitzende des Promotionsorgans: | Prof. Dr.-Ing. habil. Marion Merklein |
| Gutachter: | Prof. Dr.-Ing. Felix Freiling |
| | Dr. Pavel Gladyshev |

*Посвящается Лерочке.*

# Abstract

Standard procedures in computer forensics mainly describe the acquisition and analysis of *persistent* data, e.g., of hard drives or attached devices. However, due to the increasing storage capacity of these media and, correspondingly, significantly larger data volumes, creating forensically-sound duplicates and recovering valuable artifacts in time gets more and more challenging. Moreover, with the wide availability of free and easy-to-use encryption technologies, a growing number of individuals actively try to protect personal information against unauthorized access. If a suspect is unwilling to share the respective decryption key such measures can therefore quickly thwart an investigation. Last but not least, many sophisticated malicious applications entirely run in memory to date and do not leave any traces on hard disks anymore. Solely focusing on traditional sources can thus lead to an incomplete or inaccurate picture of an incident. In order to cope with these issues, researchers have proposed alternative investigative strategies and extracting pieces of evidence from a computer's RAM. For this purpose, a so-called memory *snapshot* is taken and inspected offline on a trusted workstation. These activities known as *memory forensics* have gained broad attention among practitioners over the last years, primarily because operations are repeatable and may be safely verified by other experts without polluting the system environment as, for instance, in a *live response* situation.

In this thesis, we give a comprehensive overview of fundamental concepts and approaches for seizing as well as examining volatile information. It consists of two parts: In the *first* part, we formalize criteria for sound memory imaging and illustrate the characteristics, benefits, and drawbacks of proven acquisition technologies available on the market to date. As we will see, especially for software-based solutions it is difficult to produce reliable memory snapshots, because the system state cannot be effectively frozen during runtime. With the help of an evaluation platform that we have developed in the course of the dissertation period, the performance and quality of software imagers can be thoroughly assessed for the first time.

In the *second* part of this thesis, we explain how common system compromise and manipulation techniques as they are typically employed by rootkits or other types of intelligent malware can be discovered during memory analysis. We also present *rkfinder*, a new plug-in for the popular, open source forensic suite *DFF* that facilitates some of these tasks. Rkfinder implements *cross-viewing* algorithms for checking the integrity of a machine and detecting possible inconsistencies that indicate the presence of a threat. By automatically highlighting suspicious resources that are likely to have been tampered with, even less experienced investigators are able to identify system areas that require particular attention. Thereby, potential sources of an intrusion can be quickly found and addressed.

# Zusammenfassung

Standardvorgehensweisen im Bereich der Computerforensik beschreiben mehrheitlich die Sicherstellung und Analyse *persistenter* Daten, zum Beispiel auf Festplatten oder extern angeschlossenen Geräten. Auf Grund der zunehmend größeren Speicherkapazität eingesetzter Medien und den damit verbundenen stetig wachsenden Datenmengen, ist eine zeitgerechte Erstellung forensisch sauberer Duplikate sowie eine anschließende Beweismitteluntersuchung mit immer mehr Schwierigkeiten verbunden. Mit der hohen Verfügbarkeit von kostenlosen und einfach zu bedienenden Verschlüsselungswerkzeugen nimmt die Zahl von Personen, die sensible Informationen bewusst gegen unberechtigten Zugriff zu schützen versuchen, weiterhin zu. Sofern ein Verdächtigter in einem solchem Fall nicht dazu bereit ist, den entsprechenden Dechiffrierungsschlüssel einem Ermittler mitzuteilen, können derartige Maßnahmen maßgeblich den Erfolg einer Untersuchung gefährden. Darüber hinaus wurden in der Vergangenheit zahlreiche Beispiele hochentwickelter Schadprogramme dokumentiert, die ihre Aktivitäten nur noch ausschließlich im Hauptspeicher eines Systems ausführen und keine Spuren auf persistenten Datenträgern mehr hinterlassen. Die Einschränkung von Untersuchungsmethoden auf traditionelle Quellen einer Computerinfektion kann daher zu einem unvollständigen oder ungenauen Bild eines Vorfalles führen.

Um den zuvor beschriebenen Missständen entgegen zu wirken, entwickelten Forscher alternative Untersuchungsstrategien und schlugen eine Beweismittelextrahierung aus den Daten des Arbeitsspeichers vor. Zu diesem Zweck muss zunächst ein so genannter *Schnappschuss* des Hauptsspeichers angelegt werden, der im nächsten Schritt zur weiteren Analyse auf eine vertrauenswürdige Arbeitsstation transferiert wird. Die im Rahmen dieser *Hauptspeicherforensik* vorgenommenen Tätigkeiten sind sowohl wiederholals auch durch andere Experten nachvollziehbar und sind deshalb als forensisch sauberer zu bewerten als entsprechende *Live*-Untersuchungen während des laufenden Betriebs.

In dieser Arbeit stellen wir die grundlegenden Konzepte und Ansätze zur Sicherstellung und Analyse *volatiler* (flüchtiger) Daten im Hauptspeicher umfassend dar. Gegenstand des *ersten* Teils ist eine Formalisierung geeigneter Kriterien für eine zuverlässige Datensicherung. Wir beschreiben ebenfalls die Eigenschaften sowie Vor- und Nachteile häufig verwendeter Akquirierungstechnologien. Wie wir weiter ausführen werden, ist insbesondere eine geeignete Abbilderstellung für softwarebasierte Lösungen schwierig, da der Systemzustand zur Laufzeit nicht effektiv eingefroren werden kann. Die Qualität dieser Werkzeuge kann mit Hilfe einer Evaluationsplattform, die durch den Autor während der Dissertationszeit entwickelt wurde, zum ersten Mal detailliert beurteilt und nachvollzogen werden.

Im *zweiten* Teil dieser Arbeit erläutern wir, wie gängige Systemkompromittierungs- und Manipulationstechniken durch Hauptspeicheranalysemethoden erkannt werden können. Derartige Techniken werden typischerweise von *Rootkits* und anderen Arten intelligenter

Schadprogramme eingesetzt. Wir stellen ebenfalls das neue Erweiterungsmodul *rkfinder* für die beliebte und frei verfügbare Forensikapplikation *DFF* vor. Rkfinder nutzt so genannte *Cross-Viewing*-Algorithmen, mit denen die Integrität des Betriebssystems überprüft sowie mögliche Systeminkonsistenzen festgestellt werden können, die auf eine Schadprogramminfektion hinweisen. Durch die automatische Hervorhebung verdächtig eingestufter Ressourcen sind selbst Ermittler mit nur geringen Erfahrungen in der Lage, kritische Systemteile zu identifizieren, die einer näheren Untersuchung bedürfen. Der Einsatz des Erweiterungsmoduls erleichtert somit die Aufspürung von Computerschädlingen in nur kurzer Zeit.

# Acknowledgments

This thesis is the result of almost 3 ½ years of research in the area of memory forensics. In the course of my dissertation period, numerous people greatly helped and supported me with my work, and I would like to express them my gratitude. First and foremost, I would like to thank my supervisor Felix Freiling for many fruitful discussions and inspirations during my time at the Computer Science Department at the University of Erlangen-Nuremberg. Without his advice and encouragement, writing this thesis would have been hardly possible. I would also very much like to thank Pavel Gladyshev from the University College Dublin for gladly accepting being my second advisor. Many thanks are generally owed to the members of the Chair for IT Security Infrastructures for providing a cheerful and friendly working atmosphere. In particular, I would like to thank Johannes Stüttgen and Thomas Schreck who never hesitated reading my research papers and giving me feedback on the individual chapters of my thesis. Johannes also spent roughly about half of the 14 months with me it took to develop the evaluation platform for memory acquisition tools presented in Chapter 4. In the uncountable hours we were searching for obscure programming errors together and digging in the mysteries of the Bochs PC emulating engine, we were still always able to joke about missing program documentation and inscrutable macro names. After all though, I think we are both happy to say we eventually brought the project to a successful end.

My diploma students Hermann Lenz and Thomas Hauenstein deserve a big thank you, too. Especially Hermann facilitated some aspects of my work by writing a great part of the *rkfinder* plug-in we will illustrate in more detail in Chapter 6. Last but not least, I am thankful for the love, patience, and understanding of my family who were there for me in harder times and whenever I needed them.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction: On Traditional and Novel Approaches in Computer Forensics

Attacks on computer systems can significantly affect the performance of an organization and may lead to substantial financial losses. In a study with 56 larger-sized U.S. companies working in different branches of the industry, the individual costs for detecting, responding to, and recovering from security incidents were estimated to comprise $8.9 million on average per year (Ponemon Institute, 2012a). According to a joint report by Detica and the Office of Cyber Security and Information Assurance, businesses that fell prey to digital industrial espionage caused annual losses of approximately £7.6 billion in the United Kingdom alone (Detica, 2011). As a different survey by the Computer Security Institute (2011) shows, about half of the interviewed 138 representative organizations in the private and public sector decided to initiate a forensic investigation once a security breach had been discovered. The main objectives of such an investigation are understanding the series of actions that were taken by the adversary, identifying and retrieving valuable pieces of evidence, and possibly bringing the issue forward to law enforcement agencies for further prosecution. Investigative methods have changed and been adapted over time though. In the following section, we will give a brief overview of major phases in the evolution process. In Section 1.1.1, we illustrate the characteristics, weaknesses, and limitations of traditional forensic approaches. More modern approaches are discussed in Section 1.1.2. Our contributions to the latter research area are subject of Section 1.2.

## 1.1 Evolution of Forensic Investigations

### 1.1.1 Traditional Investigation Approaches

Traditional approaches in computer forensics particularly focus on collecting and examining data of *persistent* storage media, e.g., of hard drives or USB-attached disks. To ensure a sound and verifiable acquisition process, procedures generally recommend "pulling the plug" of the target machine as one of the first tasks at a digital crime scene (U.S. Secret Service, 2007; U.S. Department of Justice, 2008). By explicitly cutting power to the host, unintentional data overwriting or modification, e.g., due to shutting down the operating system, are expected to be efficiently prevented. Guidelines also emphasize the importance of creating bitwise copies of each device, so-called *forensic images*, that serve as a starting point for all subsequent examinations (Stephenson, 2000; Association of Chief Police Officers, 2007). Thereby, it is possible to recover potential evidence even

in case the original medium is no longer available at a later time (Cohen, 2011). With the help of (hardware) write blockers that are used during the duplication process, the integrity of the source is protected. In turn, digital fingerprints in form of cryptographic hash sums as well as proper documentation of all steps help prove the accuracy and authenticity of the copied data and maintain a rigorous *chain of custody* that can be reprocessed and verified by other investigators.

**Weaknesses of Traditional Investigation Approaches**

Although regarded as best practices for a long time, the previously described methods have shown to struggle with a number of technical developments more recently: First, in contrast to hard drive capacities, I/O transfer bandwidths have only grown linearly (Patterson, 2004; Roussev and Richard III, 2004). Thus, generating a clone of a typical modern hard disk with a size between various hundreds of gigabyte and several terabytes can easily take multiple hours. Considering that even home users frequently possess numerous storage devices that may be relevant to a case, the data acquisition phase often needs to be extended and can delay an entire investigation. Shipley and Reeve (2006, p. 6) argue, since "the current ranks of trained computer forensics personnel are inadequate to support the ever-growing amount of digital evidence that should be collected at crime scenes[,] [i]t is fairly common for investigators to wait months for their reports". Casey (2011, p. 3) summarizes, "[e]xisting best practice guidelines are becoming untenable even in law enforcement digital forensic laboratories where growing caseloads and limited resources are combining to create a crisis". Second, while the impact of turning off ordinary desktop PCs is usually negligible, shutting down a critical company server may be overly disruptive and result in lost revenues or decreased business productivity. Moreover, depending on its complexity and configuration, removing the power cord from a running machine may severely damage system components. For instance, it is possible that file system journals are left in an inconsistent state and have to be rebuilt when the computer is started up again. Likewise, RAID arrays may be corrupted and need to be repaired. All these activities are costly, time-consuming, and may entail legal liabilities. Consequently, there is demand to "strike a balance between the requirements for a forensically sound preservation process and the business imperative of minimizing impact on normal operations" (Casey, 2010, p. 86).

In addition, there is a myriad of free and commercial software products that provide file or even full disk (FDE) encryption capabilities. Examples include the popular cross-platform *TrueCrypt* suite (TrueCrypt Foundation, 2013), *SafeGuard Easy* (Sophos Ltd., 2013), and *DiskCryptor* (Ntldr, 2013). Alternative solutions are offered by *SecurStar* (SecurStar Corporation, 2008), *Check Point* (Check Point Software Technologies Ltd., 2013), and *Symantec* (Symantec Corporation, 2013). Encryption functionality is also implemented in many modern operating systems (Saout, 2006; Microsoft Corporation, 2013a; Apple Inc., 2013), and studies have shown that organizations more and more realize the benefits of these technologies and enable the respective security features by default (Ponemon Institute, 2012b; SECUDE AG, 2012). From a forensic standpoint,

such measures can tremendously slow down or even completely stall an investigation: Standard cryptographic algorithms like *AES* are designed to render brute force attacks on a cipher code infeasible. Thus, unless a suspect is willing to share the originally chosen passphrase, evidence stored on an encrypted drive may be unrecoverable. For instance, Casey et al. (2011, pp. 130-132) illustrate cases of alleged child pornography as well as terrorism in which defendants could not be successfully convicted due to concealing incriminating information on encrypted media. For this reason, the authors conclude that "pulling the plug from a computer is not an acceptable response technique when encountering FDE or even volume encryption". Similar arguments hold true for a high percentage of malicious applications that employ different kinds of armoring, obfuscation, and self protection mechanisms in order to impede reverse engineering and static binary analysis (e.g., see Linn and Debray, 2003; Christodorescu and Jha, 2004; Young and Yung, 2004). A good overview of selected techniques used in this area and how they can be mitigated in a forensic context can be found in the work of Sikorski and Honig (2012).

It is also important to emphasize that certain worms, backdoors, or other attack programs operate solely in memory and do not leave any persistent traces any longer (Moore et al., 2003; Miller, 2004; Sparks and Butler, 2005). In a traditional investigation, these types of malware would therefore remain undetected, increasing the risk of drawing an incorrect or incomplete picture of an incident. On the other hand, there is a large number of legitimate, *portable* software packages that run entirely in RAM for convenience reasons (see Rare Ideas, 2013). Also, most major Internet browsers provide so-called *private modes* and permit opening web sites without caching the transferred data on hard disk. Although Aggarwal et al. (2010) have shown that these mechanisms can be partially defeated, reconstructing the web session of a user can take significant efforts and may require cooperation with Internet platform vendors or other third parties. Last but not least, a plethora of state- and runtime-related information, e.g., about running processes or established network connections, are lost when a computer is powered off. As Shipley and Reeve (2006, p. 7) point out, however, these sources "can, in some cases, mean the difference between solving a crime and not" or "proving someone's guilt or their innocence".

### 1.1.2 From Persistent Data-Centric Approaches to Memory-Based Investigations

To address the issues discussed in the previous section, forensic practitioners have started considering volatile information in system RAM more carefully and conducting *live* examinations of suspicious machines. A major advantage of this type of investigation is that comparatively equal amounts of evidence can be collected as during a traditional *post-mortem* analysis, yet oftentimes significantly more quickly (Prosise and Mandia, 2003; Carvey, 2007). One main drawback of this approach is that the integrity of the host is typically unknown. Specifically, when executing operating system commands,

the corresponding output may be manipulated by possibly installed malicious applications (Carrier, 2006). Because of this risk, forensic reports may be inadmissible in court, putting the value of the entire analysis at stake. Even though investigators can store a collection of trusted, statically-linked tools on read-only media to prevent such scenarios, one big problem of this methodology remains: since the state of the target computer is changed with every launched program, the workspace is increasingly polluted, and results can frequently not be independently verified and reproduced. In sum, live response measures thus heavily violate forensic principles and are generally seen as not sufficiently adequate for more delicate cases (Walters and Petroni, 2007; Waits et al., 2008). For these reasons, researchers have suggested reducing the impact on the host and only creating an image of a computer's RAM that can later be inspected offline on a secure, isolated workstation. The process of these activities is commonly referred to as *memory forensics* and has gained broad attention in the recent past, inspired by a challenge of the *Digital Forensics Research Workgroup* in 2005 (DFRWS, 2005). In this thesis, we will give an in-depth overview of the benefits and challenges of memory forensic tasks and explain techniques for acquiring and analyzing volatile information in a forensically sound manner. In the following section, we will describe our individual contributions in more detail.

## 1.2 Contributions of this Thesis

### 1.2.1 Illustration and Structuring of the Research Area

Due to the pressing need to find suitable alternatives for classic approaches, available literature in the area of memory forensics has grown rapidly in the last years. In particular, a lot of research has been done to identify crucial system structures that store core information and are of major importance for analyzing volatile information and extracting valuable pieces of evidence. However, many publications are strongly focused on specific technologies or are of highly informal nature and solely illustrate aspects of single investigative cases (for a list of selected cases, please see the *Volatility Documentation Project*, Volatile Systems, LLC, 2013b). Especially in the latter works, the boundaries under which proposed solutions can be applied frequently remain unclear though.

In this thesis, we give a comprehensive and structured overview of proven memory acquisition and analysis techniques. By illustrating the advantages, weaknesses, and limitations of the different methods, forensic practitioners have a reasonable basis for choosing an appropriate response and investigation strategy. Likewise, our insights may serve as a starting point for novice academic researchers to conduct further studies. Please note, however, that unless otherwise stated, our explanations solely refer to the family of *Microsoft Windows* operating systems. A description of memory forensic procedures for other platforms such as *Linux* or *Mac OS* is out of the scope of this thesis. Good introductory texts to these topics can be found in the works of Movall et al. (2005) and Suiche (2010) though.

### 1.2.2 Formalization of Criteria for Sound Memory Imaging

As a side effect of the rather unstructured research area, a formalization of the forensic memory acquisition and analysis process has been neglected in the past. In particular, even though investigators had frequently stressed the need for collecting volatile information in a "sound" and "reliable" manner, the meaning of these terms was only vaguely described in the literature. In this thesis, we introduce three criteria, *correctness*, *atomicity*, and *integrity*, that form the foundation for a sound and reliable memory imaging process. We derive our criteria from established theories in other respected fields of computer science. We will show that the proposed formalizations can model as well as integrate the requirements pointed out by previous authors and help determine the quality of a forensic memory snapshot more accurately.

### 1.2.3 Evaluation of Forensic Memory Acquisition Software

In order to *measure* the previously described factors, we have implemented an evaluation platform that, for the first time, offers an in-depth and repeatable testing approach for memory acquisition software. Our platform is built upon a highly customized version of the *Bochs* PC emulator (The Bochs Project, 2013a) and is designed to automatically assess the performance of the individual utilities. We have tested our solution with three popular products available on the market to date. Results showed that not all imaging applications were initially capable of generating a correct snapshot of a computer's RAM due to logical errors in their respective source code. We have developed patches for the affected program components, so that all evaluation candidates eventually duplicated volatile information successfully.

Another major observation we made in our evaluation was that the accuracy of created memory snapshots significantly depends on the level of concurrent activity on the system. Our platform allows monitoring this activity over the course of the imaging process and estimating the degree of such negative influences. Also, we are able to estimate upper bounds for the *impact* of memory imagers, i.e., the amount of memory that is overwritten by loading an acquisition program into RAM. Our insights support investigators with choosing an appropriate imaging strategy and comparing available solutions on a reasonable basis. By illustrating the benefits, but also the limitations and weaknesses of existing technologies, the decision why a particular method was selected or not can be better justified and defended in court.

### 1.2.4 Facilitation of the Memory Analysis Process for Less Experienced Investigators

Due to the increased interest in memory-based investigations, various analysis tools have been developed over the last years. However, many of these tools have originally been developed as proof-of-concept (PoC) demonstrations and are partially not well

documented. The de-facto standard in the industry at the time of this writing is the *Volatility Framework* (Volatile Systems, LLC, 2008, 2013a) we will introduce in greater detail in a later part of this thesis. The framework is very powerful and under active maintenance of an expert community, yet it mainly aims at experienced investigators that have a good knowledge of operating system structures and internals. For novice practitioners or personnel in general IT departments that lack training with respect to forensics-related tasks, the interpretation of generated reports may be too complex and demanding.

To address this issue, we present a plug-in for the well-known forensic framework *DFF* (ArxSys, 2009) that abstracts memory analysis-related aspects from the user to a significant degree. Results are correlated in the background and displayed in an intuitive and easy-to-use graphical interface. By visualizing the system state in a tree-like pane, resources of special interests, e.g., running processes, open network connections, or referenced files, can be quickly accessed and examined. Additionally, the plug-in employs several so-called *cross-viewing* techniques to check the computer for possible inconsistencies. Such inconsistencies may occur when system objects are explicitly tampered with. In many cases, manipulations of this kind indicate the presence of a security threat.

For convenience reasons, our plug-in automatically highlights inconsistent resources. Thereby, sources of a potential incident are directly visible and can be addressed within a short time. In a study that we performed in the dissertation period, we were able to detect several common *rootkits*, i.e., especially sophisticated malicious applications, found "in the wild" today. Because our plug-in operates on an existing memory snapshot during a *post-mortem* analysis, reports can be reprocessed and verified by third parties at later times, in contrast to other rootkit discovery programs that must be executed on a running machine. For these reasons, forensic guidelines like the principle of *non-interference* or the demand for *result reproducibility* (see Mocas, 2004) can be much better satisfied.

## 1.3 Outline of the Thesis

The remainder of this thesis is organized as follows: In Chapter 2, we give a comprehensive overview of the memory administration process as well as of established procedures for preserving volatile information. Criteria that determine the quality of a forensic memory snapshot are introduced and formalized in Chapter 3. The architecture of a platform for evaluating acquisition utilities is subject of Chapter 4. We also present performance results for three popular software products available on the market to date and describe weaknesses and limitations these solutions have to cope with.

The second part of this thesis deals with memory analysis- and investigation-specific aspects. In Chapter 5, we explain structured approaches for extracting pieces of evidence from a RAM image. In Chapter 6, we illustrate methods for identifying and detecting sophisticated variants of malicious software applications with memory forensic techniques.

We conclude with a short summary of our work and indicate opportunities for future research in Chapter 7.

## 1.4 List of Publications

This work is substantially based on a number of peer-reviewed articles and academic papers that appeared in scientific journals or were presented at international conferences. Unless otherwise stated, the original text corpora of these publications forms the foundation for different chapters of this thesis and was only adapted, shortened, or extended if necessary. For reasons of better readability, previously published larger text fragments by the author of this thesis will not be cited again in later sections, except for highlighting arguments of particular importance. Likewise, figures, tables, or other visual elements that were created by the author and that originally appeared in other publications will not be referenced.

In detail, the following works were created in the course of the dissertation period:

- Stefan Vömel and Felix C. Freiling: "A Survey of Main Memory Acquisition and Analysis Techniques for the Windows Operating System", published in *Digital Investigation*, Volume 8, Number 1, 2011 (Vömel and Freiling, 2011).

  This journal article was completely written by the author of this thesis, under guidance of Felix C. Freiling. The first part of the article, i.e., a description of the memory architecture and memory administration process under Microsoft Windows as well as methods for acquiring a forensic copy of a computer's RAM, is presented in Chapter 2. The second part of the article that covers analysis- and investigation-related aspects is subject of Chapter 5. Both Chapter 2 and 5 have also been extended to include recent developments and approaches in memory forensics.

- Stefan Vömel and Felix C. Freiling: "Correctness, Atomicity, and Integrity: Defining Criteria for Forensically-Sound Memory Acquisition", published in *Digital Investigation*, Volume 9, Number 2, 2012 (Vömel and Freiling, 2012).

  In this journal article, three major criteria for creating an image of volatile memory in a forensically-sound manner are proposed, formalized, and discussed. This article was also composed by the author of this thesis, advised by Felix C. Freiling. The original manuscript text has only been marginally adapted with respect to this thesis and is presented in Chapter 3.

- Stefan Vömel and Johannes Stüttgen: "An Evaluation Platform for Forensic Memory Acquisition Software", presented at the *13th Annual DFRWS Conference*, 2013 (Vömel and Stüttgen, 2013).

  In this paper, the architecture of an evaluation platform for forensic memory acquisition software is described. The platform was developed in cooperation with Johannes

Stüttgen. Precisely, Johannes designed a logging application for memory-related operations, while the author of this thesis implemented most parts of the corresponding log analyzer, developed patches for the considered acquisition programs, and wrote several helper utilities for the guest system. The instrumentation interface of the platform was designed in a joint effort.

The software evaluation was later performed by the author of this thesis. The final research paper was also written by Stefan Vömel. It forms the basis for Chapter 4, but has been extended to explain aspects of the development process of the platform in more detail.

- Thomas Hauenstein and Stefan Vömel: "Possibilities for Extracting Traces of Social Networking Sites from Volatile Memory", *in preparation*, 2013 (Hauenstein and Vömel, 2013).

  Subject of this paper is a discussion of methods and techniques for identifying and finding traces of social networking sites in memory. The manuscript text is in preparation at the time of this writing and is originally based on the diploma thesis by Thomas Hauenstein, under guidance of the author of this thesis. The corresponding proof of concept software was entirely designed by Thomas. In Chapter 5, the major research results of this work are briefly summarized.

- Stefan Vömel and Hermann Lenz: "Visualizing Indicators of Rootkit Infections in Memory Forensics", presented at the *7th International Conference on IT Security Incident Management & IT Forensics* (IMF), 2013 (Vömel and Lenz, 2013).

  In this paper, a plug-in for a popular forensic framework is presented that helps detect sophisticated system manipulation attempts. The plug-in was developed by Hermann Lenz as part of his diploma thesis. The software was evaluated together with the author of this thesis based on a number of common rootkit samples. The research paper was written entirely by Stefan Vömel. It is included in Chapter 6 of this thesis.

Last but not least, the author was involved in several other IT security- and forensics-related projects that are not or only marginally related to the content presented in this thesis (Vömel et al., 2010; Benenson et al., 2011; Dewald et al., 2013). In particular, a report of the behavior and characteristics of the German *Bundestrojaner* (federal state surveillance program) was published at a national security conference in 2012 (Dewald et al., 2012).

## Chapter 2

# Background Information

In the following, we illustrate important background information about the memory administration and acquisition process. The concepts and techniques outlined in this chapter foster a better understanding of the remaining parts of this thesis. In Section 2.1, we describe memory organization on modern computer systems and give a brief overview of the memory address space layout, the virtual-to-physical address translation process, and the paging mechanism. For reasons of simplicity, we thereby focus on the original `x86` 32-bit architecture without considering advanced memory management capabilities such as *Physical Address Extensions* (PAE) or *Address Windowing Extensions* (AWE). Readers who would like to know more about these topics are referred to the work of Russinovich et al. (2009). A detailed description of the peculiarities of the 64-bit platform can be found in the *Intel ® 64 and IA-32 Architectures Software Developer's Manual* (Intel Corporation, 2013). A short summary report of the major differences is available by Kornblum (2009). In Section 2.2, we sketch different approaches for creating a forensic image of a computer's RAM and depict their respective benefits, weaknesses, and limitations. We argue that established terms in memory forensics cannot clearly reflect the characteristics of existing acquisition methods any longer. We therefore introduce two alternative criteria that form the foundation for a basic classification matrix presented in Section 2.3. With the help of this matrix, investigators can better compare different imaging technologies and choose the solution that is most suited and applicable in a given situation.

## 2.1 Memory Administration Process

Modern multi-tasking operating systems typically do not access physical memory directly but rather operate on an abstraction called *virtual memory*. This abstraction of physical RAM requires special hardware support, i.e., the *Memory Manager* or *Memory Management Unit* (MMU), and offers several inherent advantages, for instance, the possibility of providing each process with its own, protected view on system memory or restricting read and write activities on memory areas with the help of specific privilege rules (Intel Corporation, 2013). The layout of the virtual and physical address space may differ though, and blocks of virtual memory do not necessarily need to map to contiguous physical addresses (see Figure 2.1).

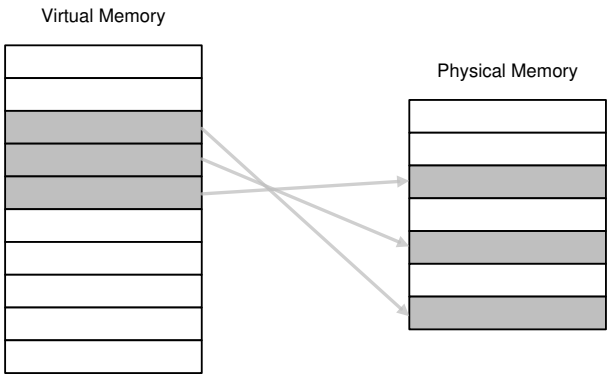



Figure 2.1: Mapping Virtual to Physical Memory Regions
(Russinovich et al., 2009, p. 14)

### 2.1.1 Memory Address Space Layout

On Microsoft Windows operating systems, each process has its own *private virtual address space* that is separated from other running applications unless portions of memory are explicitly shared. Thereby, collisions and access violations between different executables are prevented. In total, the virtual address space of a 32-bit process comprises 4 GB ($2^{32}$ bytes). Specifically, 2 GB of virtual memory are assigned to user space-related activities by default, whereas the other half of the address space, i.e., memory areas in the range of `0x80000000` to `0xFFFFFFFF`, is reserved for system usage (Russinovich et al., 2009).[1] These *kernel* regions can be jointly used by system drivers and other core components and, for example, include memory pools for dynamically-allocated data. Other major resources that are part of the system space are a number of memory maintenance structures that are required for translating virtual into physical addresses, a process we will describe in more detail in Section 2.1.2. Because system memory is a shared resource, on the other hand, kernel-level programs and artifacts must "be carefully designed and tested to ensure that they don't violate system security and cause system instability" (Russinovich et al., 2009, p. 17). In Figure 2.2, the structure of the virtual memory address space is depicted.

### 2.1.2 Virtual-to-Physical Address Translation Process

As we have already explained, processes usually operate on the virtual memory layer only. Therefore, in order to manipulate the actual data in RAM, the Memory Manager must continuously translate (*map*) virtual into physical addresses. For this purpose, memory regions are organized in separate units called *pages* at the hardware level. Each

[1] With the help of a special boot option, the user space of a *large address space-aware* process can be increased to up to 3 GB (Russinovich et al., 2009).

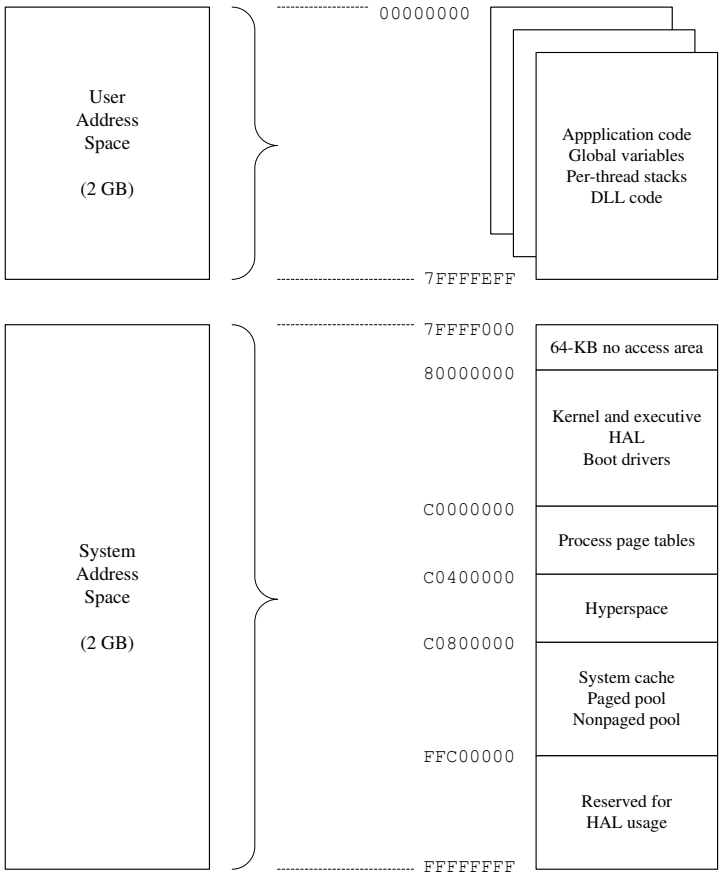



Figure 2.2: Virtual Address Space Layout
(Russinovich et al., 2009, p. 738)

page commonly has a size of 4 KB, although `x86` processors also support larger page sizes of 4 MB (Intel Corporation, 2013). To access a page, the operating system implements a multi-level approach (Russinovich et al., 2009): In the first step, a pointer to the *page directory* of the currently running process must be retrieved. The base address of this data structure is stored in the `CR3` register of the processor and is reloaded from the `_KPROCESS` block of a process at every context switch. The kernel process (`_KPROCESS`) block is part of a larger structure, i.e., the executive process (`_EPROCESS`) block, that serves as an internal representation for a Windows process. We will illustrate the `_EPROCESS` block more thoroughly in a later part of this thesis.

A page directory internally consists of 1,024 so-called *Page Directory Entries* (PDEs) that each have a size of 4 bytes. Page Directory Entries reference individual *page tables* that, in turn, contain up to 1,024 *Page Table Entries* (PTEs). A PTE is another 4-byte data structure that points to a specific page in memory. Thus, once the page directory of a process has been determined, a virtual address can be translated into its physical

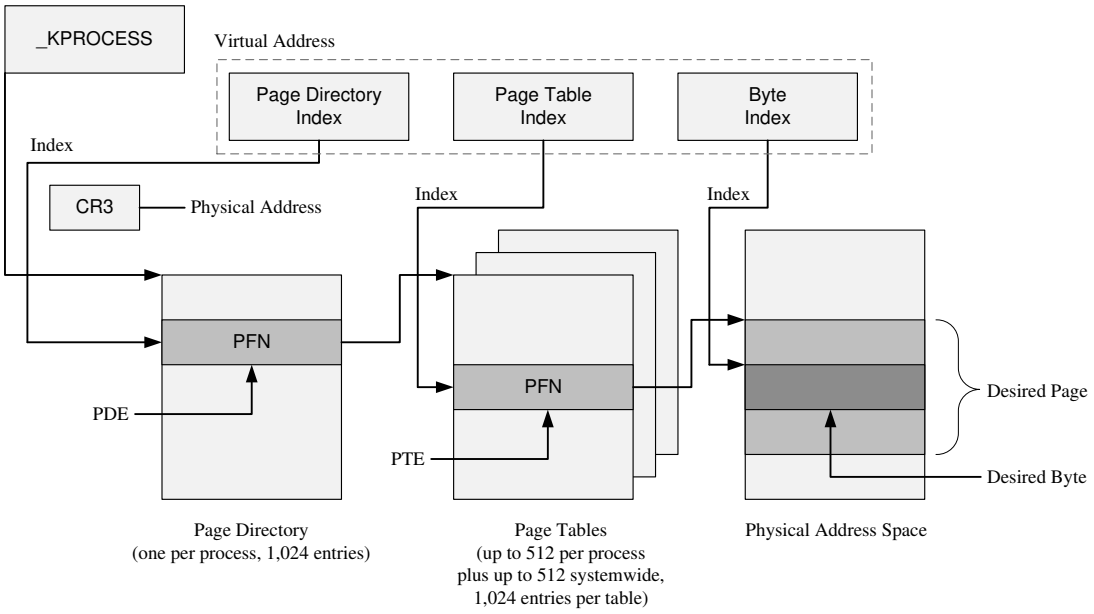



Figure 2.3: Virtual-to-Physical Address Translation Process
(Russinovich et al., 2009, p. 763)

counterpart as follows (Russinovich et al., 2009): The first 10 bits of the virtual address serve as an index into the page directory and specify the PDE in question. With the help of the PDE and the *page table index*, i.e., the subsequent 10 bits of the virtual address, the page table and corresponding PTE are identified in the next step. By parsing the PTE as well as the 12-bit *byte index* of the virtual address, the requested data in RAM can eventually be found. A summary of this process is illustrated in Figure 2.3.

### 2.1.3 Paging

With respect to the address translation process outlined in the previous section, we have implicitly assumed that the requested information is always available in main memory. However, in some cases, the total amount of virtual memory that is consumed by running programs exceeds the capacity of the entire physical storage. To cope with these scenarios, the operating system can temporarily swap out (*page*) memory contents to hard disk. Thereby, portions of RAM are freed and can be safely used by other applications. When a thread attempts to access a swapped-out page at a later time, the memory management unit generates a *page fault*, and the requested information is transparently transferred back into memory. As these operations are abstracted from the user, a virtual address space that is much larger than the size of available physical memory can be simulated. Whether or not data has been paged to disk is indicated by a special flag in the PTE (Russinovich et al., 2009). If the least significant bit of an entry is set to `1`, it is

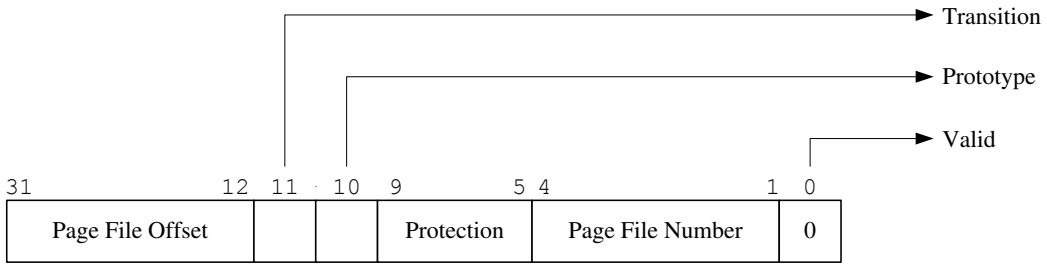



Figure 2.4: Example of an Invalid Page Table Entry (PTE)
(Russinovich et al., 2009, p. 775)

regarded as *valid*, and the corresponding page is correctly accessible in memory. In contrast, when the flag is cleared and both the *Transition* (`11`) and *Prototype* (`10`) bits are set to `0`, the entry points to an offset in a *page file* on the secondary storage of the system. Windows supports up to 16 different page files, each having a size of up to 4,095 MB (Hameed, 2007). The current file in use is denoted by a special field in a PTE, i.e., the *Page File Number* (PFN, bits `1` to `4`, see Figure 2.4). By default, the PFN references the standard page file `pagefile.sys` that is saved in the root directory on the partition the operating system is installed on. However, the names and locations of the files can be easily adapted and specified by editing the `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles` key in the Microsoft Windows registry.

Please note that we have only given a brief description of the flags and elements contained in a PTE. A more detailed explanation of the individual components and attributes can be found in the work of other authors (Kornblum, 2007; Savoldi and Gubian, 2008). The concepts presented in this section need to be thoroughly understood though, because they form the basis for many memory acquisition approaches and tools. An overview of proven methods for creating a forensic memory snapshot of a computer's RAM is subject of the following section.

## 2.2 Approaches and Techniques for Forensic Memory Acquisition

Techniques for capturing volatile data have conventionally been divided into *hardware*- and *software*-based solutions in the literature (e.g., see Vidas, 2006; Maclean, 2006; Garcia, 2007). While the latter ones depend on functions provided by the operating system, hardware-based approaches directly access a computer's memory during the imaging process and, therefore, have long been regarded as being more secure and reliable. A publication by Rutkowska (2007) has indicated, however, that these assumptions no longer hold true. Moreover, several concepts that have been presented in the recent past rely on a combination of both hardware and software mechanisms and cannot be clearly categorized with the existing terminology (Libster and Kornblum, 2008; Hal-

derman et al., 2008; Vidas, 2010). For these reasons, we believe that a classification solely on implementation-specific attributes is obsolete and is not capable of properly characterizing the latest developments any more.

As a viable alternative, Schatz (2007a,b) has proposed rather examining the conditions under which specific solutions can operate best and assess existing technologies with respect to potential influencing factors that may affect the creation of a (sound) memory copy. Two major variables Schatz has identified are the *atomicity* and *availability* of a technique. The demand for *availability* stipulates that an imaging method must be "working on arbitrary computers (or devices)" (Schatz, 2007a, p. S128). More precisely, an acquisition approach that is characterized by a high availability does not make any assumptions about particular, pre-incident preparatory measures or pre-configurations and can be applied even without detailed knowledge of the respective execution environment. Thereby, the solution is suited for most scenarios, including incident response situations.

*Atomicity*, on the other hand, intuitively reflects the demand to produce an accurate and consistent image of a host's volatile storage. Concurrently running processes in the course of the imaging period may disturb this process and result "in the memory image being imprecise, and not attributable to a specific point in time" (Schatz, 2007a, p. S128). Because these properties are contradictory to classic perceptions of *forensic soundness*, methods that satisfy higher degrees of atomicity are generally preferable to less atomic approaches when in doubt.

For the remainder of this chapter, a rough understanding of the previously described terms is sufficient. Based on our explanations, we will outline the benefits, limitations, and drawbacks of existing memory acquisition solutions in the following sections. In Chapter 3, we will define and formalize the criterion of atomicity as well as other factors for sound memory imaging in greater depth.

### 2.2.1 Memory Acquisition Using a Dedicated Hardware Card

One of the first propositions for obtaining a forensic image of a computer's RAM described the use of a special hardware card. In 2004, Carrier and Grand presented a proof-of-concept (PoC) solution called "Tribble" that duplicates areas of physical memory via *Direct Memory Access* (DMA). Because the approach is independent from possibly subverted functions of the operating system, its reliability is assumed to be high. Tribble is designed as a dedicated PCI device and is capable of saving volatile information to an attached storage medium. Upon pressing an external switch, the card is activated, and the imaging procedure is initiated. During this process, the CPU of the target host is temporarily suspended to prevent an attacker from executing malicious code and illegitimately modifying the status of the system. Once all operations are completed, control is given back to the operating system, and the acquisition card returns to an idle state again. Two lesser known but comparable implementations are suggested by

Petroni et al. (2004) with their "Copilot" prototype and BBN Technologies (2006) in the form of "FRED", i.e., the *Forensic RAM Extraction Device*.

In comparison to existing solutions at that time, the described methods offer several inherent advantages: First, as the processor of the target machine is successfully halted, the imaging operation can be atomically completed without interference by other processes. Second, because all information is directly retrieved from physical RAM, the procedure has long been believed to act outside the view of even sophisticated malicious applications such as rootkits, resulting in a pristine copy of a computer's memory (Kornblum, 2006). In a more recent presentation that gained broad attention, Rutkowska (2007) has pointed out though that the memory map of the Northbridge can be subject to manipulation (see also Sang et al., 2011). Thereby, it is possible to present a different view of physical memory to peripheral devices. Due to these findings, several authors have stated that hardware cards can no longer be fully trusted and must not be regarded as forensically sound any more, making the development of more robust and reliable memory acquisition techniques necessary (Libster and Kornblum, 2008; Ruff, 2008). In addition to these concerns, it is important to emphasize that a PCI card must be installed *prior* to its use and, thus, is only suited for certain scenarios. Specifically, Carrier and Grand (2004, p. 12) clarify that "the device has not been designed for an incident response team member to carry in his toolkit", but "rather needs to be considered as part of a forensic readiness plan". According to the authors, a card is therefore most beneficial when it is built-in in (business-)critical servers "where an attack is likely and a high-stake intrusion investigation might occur". Other options for deployment are, for instance, within a *honeypot* environment.[2] In this case, volatile information can be comfortably captured before a decoy is shut down and set up again.

### 2.2.2 Memory Acquisition via a Special Hardware Bus

As an alternative to PCI cards, Dornseif and Becher (2004) as well as Becher et al. (2005) have suggested reading memory via the IEEE 1394 (*FireWire*) bus as early as in 2004. Several years later, Gladyshev and Almansoori (2010) presented their *Goldfish* project and illustrated how volatile information can be acquired on a Mac OS system in the course of a digital investigation. A similar set of tools for the Linux platform was provided by Piegdon and Pimenidis (2007). With respect to the product family of Microsoft-based operating systems, Boileau (2006b, 2008) was first in demonstrating the feasibility of the approach for Microsoft Windows XP. In turn, Panholzer (2008) and Böck (2009) succeeded in accessing memory over FireWire in Windows Vista and Windows 7, respectively.

Imaging activities do not necessarily need to be restricted to the IEEE 1394 interface though, but "any hardware bus can potentially be used" (Ruff, 2008, p. 84). For example,

[2] A honeypot is "an information system resource whose value lies in unauthorized or illicit use of this resource" (Spitzner, 2003b,a). It acts as an electronic decoy for studying the behavior of (Internet) miscreants. More information on this topic can be found in the work of the Honeynet Project (2004).

Direct Memory Access (DMA) operations can be performed both over the PCMCIA (*PC-Card*) bus (Hulton, 2006) or over the more modern *Thunderbolt* port (Maartmann-Moe, 2012). With regard to the latter method, a working prototype has been released by the same author (Maartmann-Moe, 2013).

Retrieving volatile information via a hardware bus can address some of the issues we have outlined in the previous section. For instance, the FireWire interface is available in a large number of systems, especially laptops. Consequently, the method is suited even for scenarios that permit little or no time for pre-incident preparation. On the other hand, as Vidstrom (2006) notes, random system crashes or similar stability issues must be expected when accessing regions in the *Upper Memory Area* (UMA), i.e., addresses in the range from `0xA0000` (640 KB) to `0xFFFFF` (1 MB) (see also Zhang et al., 2011). In addition, Carvey (2007) and Boileau (2006a) have indicated inconsistencies after comparing created images with raw memory dumps. Last but not least, because the size of a host memory address is limited to 32 bits, a maximum of 4 GB of memory can be referenced (Promoters of the 1394 Open HCI, 2010). Taking these aspects into consideration, obtaining a copy of a computer's RAM via a hardware bus can therefore not be seen as sufficiently reliable at the time of this writing.

### 2.2.3 Memory Acquisition with the Help of Virtualization

The concept of *virtualization* permits simulating complete, isolated, and reliable system environments, so-called *virtual machines*, on top of a host computer (Smith and Nair, 2005). A special software layer, the *virtual machine monitor* (VMM), is responsible for sharing as well as managing and restricting access to the available hardware resources. By emulating replicas of the different physical components, each virtual machine is equipped with its own virtual processor, memory, graphics adapter, network and I/O interface, and may run in parallel to other *guest systems*. One exceptional characteristic of a virtual machine is its capability to be suspended, i.e., to pause its execution process. In this case, the state of the guest operating system is temporarily frozen, and its main memory is saved to a file on the hard disk of the underlying host. For instance, when suspending an instance of a *VMware*-based machine, a `.vmem` file, located in the working directory of the virtual machine, is created (VMware, Inc., 2013). This file contains a copy of the volatile storage in raw format and can simply be duplicated.

Virtual machines were originally believed to play only a minor role in digital investigations. For example, Carvey (2007, p. 95) pointed out that "virtualization technologies do not seem to be widely used in systems that require the attention of a first responder". With the growing importance of Internet-hosted services, this situation has drastically changed over time though, and forensic analysts are likely to encounter running virtual machines in practice. With respect to these scenarios, a physical memory snapshot of a guest system can then be acquired with only little effort.

In addition to the software-based technique outlined above, Martignoni et al. (2010) as well as Yu et al. (2012) have presented two approaches that rely on the recent *VMX* in-

struction set of the Intel architecture (Intel Corporation, 2013). Thereby, the acquisition program acts as a minimal virtual machine monitor and transparently transforms the target operating system into a virtualized guest. Because the solution essentially runs in *ring -1* privilege mode, i.e., beneath the kernel level, it is immune to attacks originating from the host, and the contents of memory can be safely and atomically duplicated. On the other hand, several authors have proven that the hypervisor layer may be subject to manipulation, too (King et al., 2006; Rutkowska, 2006; Zovi, 2006; Myers and Youndt, 2007). To cope with these issues, Wang et al. (2011a) and Reina et al. (2012) have suggested running acquisition-related operations on the *firmware level* by leveraging the System Management Mode (SMM). The System Management Mode "offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications" (Intel Corporation, 2013, p. 1025). Due to these characteristics, it is possible to generate an atomic image of memory as well as save the contents of relevant CPU registers. A major disadvantage of this method is, however, that it requires patching the BIOS of the target machine. Because the system subsequently needs to be rebooted to make the changes effective, the technique is restricted to specific scenarios in controlled environments.

### 2.2.4 Memory Acquisition Using Software Crash Dumps

Starting with Microsoft Windows 2000, all modern versions of the Microsoft Windows product family can be configured to save important debugging information to hard disk when the target machine unexpectedly stops working (Microsoft Corporation, 2012a). The execution of the operating system can also be intentionally interrupted by pressing the `ScrollLock` key twice, while holding the right control (`Ctrl`) key at the same time (Microsoft Corporation, 2011). When the keyboard combination is invoked, the internal `KeBugCheckEx` function is executed with the stop code `0xE2` that signals the initiation of a "manual crash". As a result, interrupts on all processors of the system are disabled, a blue screen is displayed, and a so-called *crash dump file* is generated by overwriting parts of the page file. For a more detailed description of this process, the reader is referred to Russinovich et al. (2009).

In contrast to many other acquisition approaches, a created crash dump does not only contain volatile information from main memory but also of relevant processor registers. With the help of these registers, valuable artifacts such as the Page Directory of a process (see Section 2.1.2) can be quickly restored in the course of a memory-based investigation. It is important to note, however, that a dump file does not necessarily include a copy of the *entire* physical address space but may comprise smaller subsets, in dependence of the system configuration. For instance, a *kernel memory dump* solely contains areas of the system space, while a *minidump* only saves memory regions that are related to the currently running process. The option for acquiring a specific snapshot type can be adapted in the Control Panel of the operating system, a list of standard values for different platforms is maintained in the Microsoft Knowledge Base (Microsoft Corporation, 2012a).

One major drawback of the crash dump method is that the previously described keyboard shortcut is disabled by default. In order to activate it, an investigator must manually edit the `CrashOnCtrlScroll` value in the Microsoft Windows registry and subsequently reboot the system (Microsoft Corporation, 2013d). Due to these requirements, the technique is of limited use in situations where immediate actions need to be taken. On the other hand, in a controlled environment, the approach permits obtaining a snapshot of physical memory with a comparatively high degree of atomicity. Furthermore, even though the file format of a crash dump is proprietary, it has been well studied in the past (Schuster, 2006a, 2008a) and can be natively processed by both the Microsoft Windows kernel debugger as well as standard memory analysis frameworks today. Given sufficient time for preparation, the solution is thus well applicable in practice.

### 2.2.5 Memory Acquisition with the Help of Software Imagers

One of the most widely used techniques for obtaining a snapshot of a computer's RAM involves the use of a special imaging application. Classic tools accessed physical memory directly in user space (Garcia, 2007; Ruff, 2008). For security reasons, however, these capabilities were revoked with the introduction of the first service pack for Microsoft Windows Server 2003 (Microsoft Corporation, 2013b) and are generally not available in modern operating systems any longer. To cope with these restrictions, most software vendors ship their products with a kernel-level driver to date. The driver typically opens a handle to the internal `\\.\Device\PhysicalMemory` section object and sequentially maps the address space for later duplication. We will illustrate this process in more detail in Chapter 4 when we present our evaluation platform for forensic memory acquisition software.

Investigators may choose between a myriad of free and commercial solutions being available on the market to date: Well-known utilities include, for example, *KnTDD* (GMG Systems, Inc., 2007), *mdd* (ManTech CSI, Inc., 2009), *Memoryze* (Mandiant, 2011), *FTK Imager* (AccessData, 2012), *WinPMEM* (Cohen, 2012b), *FastDump* (HBGary, 2013), or *MoonSols* (Suiche, 2013). The functionality of these tools is frequently similar, and we will evaluate the performance of selected applications in a later part of this thesis. In addition to the noted programs, several other tools permit acquiring the address space of a single, specific process. For example, with *PMDump*, it is possible to "dump the memory contents of a process to a file" (Vidstrom, 2002). A comparable solution is provided by Klein (2006b) in the form of *Process Dumper* that redirects all collected information to standard output by default. Thus, the final image can be easily transferred over a remote connection for further investigation as well, e.g., with a simple network administration utility such as *netcat* (nc).[3] However, both PMDump and Process Dumper also have various drawbacks: First, since the applications are closed source and use a

[3] Netcat does not establish an encrypted communication channel by default. To securely transfer data over the network, the connection can be tunneled over the SSH (*Secure Shell*) protocol. Alternatively, the *cryptcat* implementation may be used as well. For more information on these topics, please see Farmer and Venema (2005a).

proprietary data format, verifying their correctness or adapting their functionality is difficult. Second, because an explicit process ID must be passed as a parameter when launching the programs, a corresponding process listing utility must be run in the first place. Thereby, the level of system contamination on the target host is further increased though.

The previous arguments can be generalized for all software-based imagers: In order to create a copy of physical memory, the respective solution must be first loaded into RAM. As a consequence, portions of memory are irrevocably overwritten, and valuable pieces of evidence may be destroyed. Sutherland et al. (2008) have shown in a study that the impact of a tool on the target host can be significant. Furthermore, in the course of the imaging period, concurrently running processes may access and modify the address space as well. As we have already argued, such operations affect the accuracy and consistency of the generated snapshot and are therefore contradictory to classic perceptions of forensic soundness. Last but not least, it is also crucial to point out that the described methods inherently rely on functions provided by the operation system and, as such, are vulnerable to attacks. For instance, a malicious program can block direct access to the `\\.\Device\PhysicalMemory` section object and, thus, prevent memory imaging completely (see Crazylord, 2002). Although these types of manipulations are fairly easy to identify and immediately indicate the presence of a threat (Kornblum, 2006), several much more sophisticated malware species are known that may be even difficult to discover for trained practitioners (Sparks and Butler, 2005; Bilby, 2006).

In sum, software-based imaging must be critically judged from a forensic standpoint: On the one hand, most applications do not require special hardware setups or system configurations. For this reason, they are generally suitable for most incident scenarios and permit capturing a forensic image even in case a first responder only has little time for preparation. On the other hand, trusting the output of an unknown system eventually "decreases the reliability of the evidence" (Carrier and Grand, 2004, p. 6) and may put an entire investigation at stake. Forensic analysts should be aware of these peculiarities and should carefully decide in what cases – and what not – using an acquisition utility is appropriate.

### 2.2.6 Memory Acquisition via Operating System Injection or Adaption

Schatz (2007a) has presented a proof-of-concept demonstration called *BodySnatcher* which injects an independent operating system into the possibly subverted kernel of a target machine. By freezing the state of the host computer and solely relying on functions provided by the acquisition OS, an atomic and reliable snapshot of the physical address space may be created. The presented prototype suffers from several inherent drawbacks though: First, it is platform-specific to a high degree due to its low level approach and high complexity. Second, injecting the alternative operating system changes significant amounts of memory, similarly to the software-based technologies outlined in the previous section. What is worse, BodySnatcher only supports data operations over

the very slow serial port. According to Schatz, creating an image of 128 MB of memory takes about 45 minutes with a speed of 115 kbps. With respect to modern computer systems and considerably higher RAM capacities, these benchmarks are unacceptable. Consequently, even though the concept is promising, its technical constraints prevent it from being truly applicable in real-world situations.

In contrast, Libster and Kornblum (2008) have recommended integrating acquisition-related program logic as a separate module into the system core. This module is loaded at boot time and can be invoked by a special keyboard trigger. In this case, currently running processes are suspended to ensure atomic imaging operations. The authors also encourage implementing support for several storage dump locations, e.g., remote network drives or externally attached media. For increased security, the use of hardware-side, read-only memory flags or encrypted Trusted Platform Modules (TPMs) is suggested. These mechanisms shall guarantee the integrity of the imaging process and prevent attacks on the executing code.

The proposal made by Libster and Kornblum is viable and addresses some of the issues previously described acquisition methods are confronted with to date. As we have argued in an earlier publication though, major concerns that may be expressed by platform vendors are that "an atomically-operating acquisition module that is embedded in the system kernel is in blatant contrast to operating system guidelines that stipulate multi-parallelism and generally try to prevent resource monopolization" (Vömel and Freiling, 2012, p. 136). Therefore, the approach has yet to be realized in practice.

### 2.2.7 Memory Acquisition via Cold Booting

Halderman et al. (2008) have illustrated a memory acquisition approach that leverages the so-called *remanence effect* of RAM modules. The method is based on the observation that, in contrast to prior assumptions, volatile information is not immediately erased when a machine is powered off but is still retained for short periods of time, even at normal room temperatures (Anderson, 2001; Gutmann, 2001). Chow et al. (2005) indicated in a study that these effects can be noticed as well after a computer has been rebooted. Further experiments have shown that data remanence times can be significantly extended by artificially cooling down the individual RAM chips. For this task, standard refrigerants are perfectly viable. Halderman et al. (2008, p. 2) note that at temperatures of approximately −50°C, "decay rates were low enough that an attacker who cut power for 60 seconds would recover 99.9% of bits correctly". With the help of liquid nitrogen or other liquefied gases, the time span can be substantially prolonged, leading to decay rates "of only 0.17% after 60 minutes". The authors therefore argue that "even in modern memory modules, data may be recoverable for hours or days with sufficient cooling" (Halderman et al., 2008, p 5).

Due to the characteristics outlined above, forensic analysts have the possibility of invoking a *cold boot attack* on a target computer and, thereby, obtain a copy of its physical memory. For this purpose, power to the host is briefly cut. In the subsequent reboot

phase, a custom kernel is started that launches a minimal acquisition program and initiates the imaging process. The usability of this approach has been proven in a number of recent works: Vidas (2010) has demonstrated the proof-of-concept utility *AfterLife* that saves the contents of physical RAM to an external storage medium after rebooting. The implementation depicted by Chan et al. (2008, 2009) provides an investigator with an interactive shell when the computer is restarted and permits analyzing state-related data on the fly. Both projects are still in development though and are not sufficiently mature yet for daily use.

An alternative, even more powerful technique involves installing the different RAM modules in another system that is under control of the analyst. As Halderman et al. (2008, p. 2) point out, these measures "deprive the original BIOS and PC hardware of any chance to clear the memory on boot". Carbone et al. (2011) criticize, however, that this method may bear risks for particularly inexperienced practitioners, because the cooling liquids or gases that are required for safely transplanting the chips frequently contain toxic or inflammable components. Thus, all operations must be performed with great care.

### 2.2.8 Memory Acquisition Using the Hibernation File

The Windows hibernation file may be an important source for a forensic investigator, too. Similarly to a *crash dump file* (see Section 2.2.4), it does not only contain a copy of physical memory, but also the contents of relevant processor registers. When a computer is about to be suspended to disk and transitions to a so-called *S4* mode (ACPI Promoters Corporation, 2011)[4], the system state is temporarily frozen, and a compressed RAM snapshot is created and saved to the `hiberfil.sys` file in the root directory of the system partition (Russinovich et al., 2009). Even though the file format is proprietary, it has been successfully reverse engineered in the past (Suiche, 2008) and is supported by standard memory analysis frameworks to date. In addition, with the help of the *MoonSols* suite (Suiche, 2013, see Section 2.2.5), it is possible to convert the `hiberfil.sys` file into a raw memory dump to facilitate data extraction and analysis even further.

From a forensic standpoint, the hibernation file may yield valuable pieces of evidence. However, Carvey (2007, p. 96) also argues that it may possibly be significantly out of date in dependence of the user behavior and in "most cases[,] will not contain the current contents of memory". Investigators should keep these aspects in mind when conducting their examinations.

## 2.3 Categorization of Forensic Memory Acquisition Approaches

Based on the two factors *availability* and *atomicity* we have introduced in Section 2.2, we are now able to broadly group and compare the previously illustrated memory acqui-

[4] The *Advanced Configuration and Power Interface* (ACPI) specification is jointly developed by the Hewlett-Packard, Intel, Microsoft, and Toshiba Corporation as well as Phoenix Technologies Ltd.

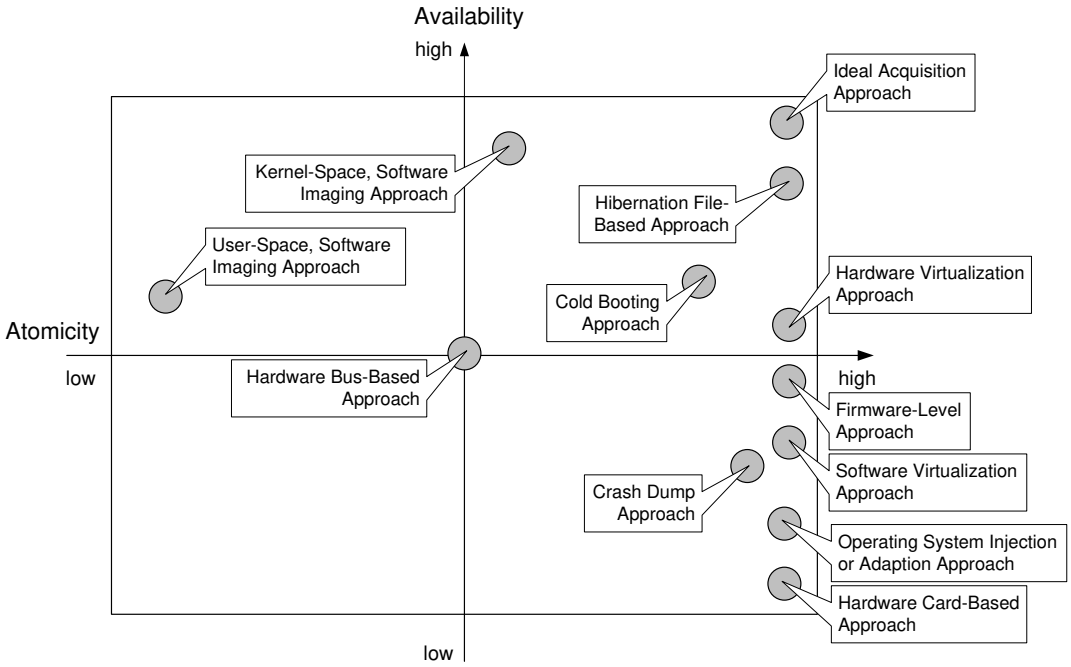



Figure 2.5: Categorization of Memory Acquisition Approaches
(Based on Schatz, 2007b)

sition approaches. The corresponding decision matrix is visualized in Figure 2.5. Please note, however, that the exact positioning of the individual methods within the fields of the matrix may certainly be subject to discussion in parts. Likewise, the discussed dimensions do not describe the characteristics of existing technologies in their entirety. In Chapter 3, we will formalize and describe criteria for sound memory imaging in more detail. In spite of these issues, we believe that the presented model serves as a starting point for giving investigators a good insight into when – or when not – to choose a specific solution. Vital points to keep in mind are especially:

- An ideal acquisition method is characterized by both a high degree of atomicity and availability and is therefore located in the right upper corner of the matrix.
- Techniques that are listed in the right half of the matrix must generally be favored upon techniques that are grouped on the left side, because they are better capable of coping with concurrent activity (and are possibly more available as well).
- Methods that are located in the bottom field on the left side of the matrix (currently: none) are generally not suitable for obtaining volatile information in a forensically sound manner and must not be considered further.
- Approaches that are categorized in the right bottom field of the matrix are applicable for scenarios where an investigator has sufficient time for pre-incident preparation.

- Techniques that are listed in the right upper field of the matrix are especially suited for *smoking gun* situations, i.e., where little time between an incident and the investigation phase has passed.

## 2.4 Summary

In this chapter, we have given a general overview of memory administration on the `x86` 32-bit platform. Additionally, we have illustrated the layout of the virtual address space, the virtual-to-physical address translation process, and the paging mechanism as implemented in the product family of Microsoft Windows operating systems. These concepts must be thoroughly understood, because they form the foundation for many memory analysis techniques presented in later parts of this thesis. We have also described various approaches for creating a copy of a computer's RAM and explained their individual benefits, drawbacks, and limitations. These approaches partially differ tremendously with respect to their availability and their capability of coping with concurrent activity. For instance, the use of special hardware cards permits acquiring the contents of memory with a comparatively high degree of atomicity. However, these cards must be built in prior to an incident and are therefore only valuable in specific scenarios. Software-based imaging solutions, on the other hand, are applicable in most forensics-related situations, but are only able to create a "fuzzy" snapshot that is not attributable to a single point of time. Moreover, these technologies depend on functions provided by the host operating system and, thus, are especially vulnerable to potentially installed malicious applications. Investigators must keep these aspects in mind when balancing the need for availability with the reliability of a particular method.

In order to support their decision strategy, we have outlined a classification matrix that groups existing technologies in distinct fields. Thereby, analysts may quickly choose an approach that is best suited for a case. In the following chapter, we will define and formalize criteria for sound memory imaging in more detail. With the help of these criteria, the quality of memory acquisition solutions can be better determined and assessed.

# Chapter 3

# Criteria for Sound Memory Acquisition

As we have indicated in the introduction of this thesis, memory forensics has moved more gradually into the focus of security professionals over the last years and is increasingly regarded as an integral part of an investigation. Especially the evidence extraction and examination phase we will describe in Chapter 5 has been thoroughly covered in the literature and has received broad attention by both academic researchers as well as forensic practitioners. With respect to the analysis process, authors frequently request the corresponding memory snapshot to be "sound" or "reliable" (e.g., see Maclean, 2006; Garcia, 2007). However, even though we have outlined the characteristics of existing memory acquisition approaches in the previous chapter, an overall evaluation of these techniques is mostly missing. What is worse, properly defining the meaning of forensic soundness and reliability in the context of memory forensics has been utterly neglected in the past and still remains vague and informal to a high degree. In order to address these issues, we will introduce and formalize three fundamental criteria, *correctness*, *atomicity*, and *integrity*, in this chapter. We will show that with the help of these criteria, the quality of a memory snapshot can be determined. Thereby, we can set a starting point for actually *measuring*, instead of estimating, the performance of acquisition technologies, in contrast to many previous works that merely explain specific behavioral aspects.

### Outline of the Chapter

The remainder of this chapter is outlined as follows: In Section 3.1, we give a short overview of existing memory acquisition models and describe their benefits and limitations. In Section 3.2, we illustrate several terms originally used in distributed systems theory. Based on these terms, we define our criteria for forensically-sound memory imaging. Formalizing the definition of *correctness*, *atomicity*, and *integrity* of a memory snapshot is subject of Section 3.3. In Section 3.4, we discuss the meaning of forensic soundness in the context of memory-based investigations, followed by a description of how previously established terms in memory forensics can be mapped and integrated into our model in Section 3.5. Challenges that memory acquisition solutions must frequently cope with in practice are discussed in Section 3.6. We conclude with a short summary of our findings and a brief outlook on the capabilities our proposed model permits in Section 3.7.

## 3.1 Overview of Existing Memory Acquisition Models

Even though the memory acquisition process has been researched to a lesser degree, several authors have discussed different factors that affect the quality of a forensic memory snapshot in the past. Schatz (2007a) was first in identifying three major criteria for forensically-sound memory imaging, namely the *fidelity* and *reliability* of the memory copy as well as the *availability* of the respective acquisition method. The principle of *fidelity* dictates that the generated memory image is "a precise copy [of] the original host's memory" (Schatz, 2007a, p. S128). On the other hand, *reliability* stipulates that an acquisition technique is not vulnerable to subversion and either produces "a trustworthy result or none at all". Last but not least, *availability* refers to the applicability of a method "on arbitrary computers (or devices)".

As we have indicated in the previous chapter, Schatz (2007b) adapted these criteria in a later work and outlined a preliminary evaluation framework for memory acquisition techniques based on the two dimensions *atomicity* – which serves as a metric for fidelity – and *availability*. However, both dimensions were only vaguely described, and especially the definition of atomicity remained unclear. In this chapter, we will elaborate the characteristics of an atomic memory image in depth and derive suitable metrics for assessing the performance of acquisition solutions. We will also show that the atomicity of an approach must be determined in independence from other fundamental influencing variables, for instance, its correctness. Thereby, differences between these factors will be illustrated based on a number of intuitive examples.

As an alternative to the criteria outlined by Schatz, Inoue et al. (2011b) described four metrics in a study, namely the *correctness* of a method, its *completeness*, *speed*, and the *amount of interference*. Precisely, *correctness* demands that "the physical address of a page in the image [corresponds to] the actual physical address of that page in memory" (Inoue et al., 2011b, p. S43). In turn, the notions of *completeness* and *speed* request that the snapshot contains "all of the physical address space which is not allocated to devices or the BIOS" and is recorded "as quickly as possible". Finally, the *amount of interference* refers to "the amount of memory [a method] alters on the machine" and should be minimized. As we will see, the proposed criteria come close to the definitions we suggest in this chapter. However, we will also show that our criteria are capable of mapping and integrating the original metrics and permit greater flexibility as well as easier adaptability when assessing memory acquisition solutions.

## 3.2 Background on Distributed Systems

In the following, we will briefly describe major characteristics of distributed systems. The explanations given in this section are mainly based on the works of Lamport (1978) as well as Mattern (1989) and need to be thoroughly understood, because they form the foundation for our memory acquisition model presented in Section 3.3.

### 3.2.1 Characteristics of Distributed Systems

A distributed system can be viewed as a collection of distinct processes that cooperate and communicate with each other by exchanging messages (Lamport, 1978). Thereby, a *process* in a distributed system is characterized by a sequence of individual *events*. Such an event is triggered when a message is sent to a process, a message is received by another process and, as a consequence, the internal state of the recipient is updated, or said state is changed by performing a local computation. The events of a single process are totally ordered according to their local occurrence. In more detail, for two local events $a$ and $b$, it is possible to introduce a "happened before" relationship, denoted by $a \rightarrow b$, if $a$ *precedes* $b$. In contrast, if $a$ does *not* precede $b$, we denote this by $a \nrightarrow b$. With respect to the entire system, this relationship can be extended as follows (see Lamport, 1978, p. 559):

1. For two events $a$ and $b$, if $a$ is the sending event of a message and $b$ is the corresponding receive event, then $a \rightarrow b$.
2. Given three events $a$, $b$, and $c$, the relation is *transitive*, i.e., if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

As can be seen, the previously described relation is unidirectional, and an event $a$ that "happens before" an event $b$ can *causally* affect this event but not vice versa. According to Mattern (1989, p. 121), it is this causality relation that is "the central concept [...] which determines the primary characteristic of time, namely that the future cannot influence the past". Likewise, two events $a$ and $b$ are *concurrent* when they do not causally affect each other, i.e., $a \nrightarrow b$ and $b \nrightarrow a$.

The flow of messages and events may be graphically visualized with the help of a *space-time diagram* as shown in Figure 3.1. In the example, the vertical direction of the diagram represents the space of distinct processes, whereas the horizontal direction represents (real) time. Events that occur at a specific point of time are indicated by a single dot, the exchange of messages is depicted by diagonal arrows. An event $a$ that "happens before" (causally affects) an event $b$ is horizontally aligned *to the left* of this event. As such, by following the respective arrows "in time" from the left to the right, it is easy to see that $e_{11} \rightarrow e_{21}$ and $e_{31} \rightarrow e_{22}$. Likewise, $e_{31} \rightarrow e_{33}$, due to the transitivity condition outlined above. However, $e_{22} \nrightarrow e_{32}$, and $e_{32} \nrightarrow e_{22}$, thus, $e_{22}$ and $e_{32}$ are concurrent. Please note that, for instance, the events $e_{21}$ and $e_{32}$ are *also* concurrent, even though, as depicted in the space-time diagram, the event $e_{21}$ appears to happen at an earlier time than the event $e_{32}$. However, since the events are not causally related, process $p_3$ is only informed about the state of process $p_2$ after receiving the corresponding message in event $e_{33}$. By the same reasoning, the events $e_{31}$ and $e_{21}$, for instance, are also concurrent.

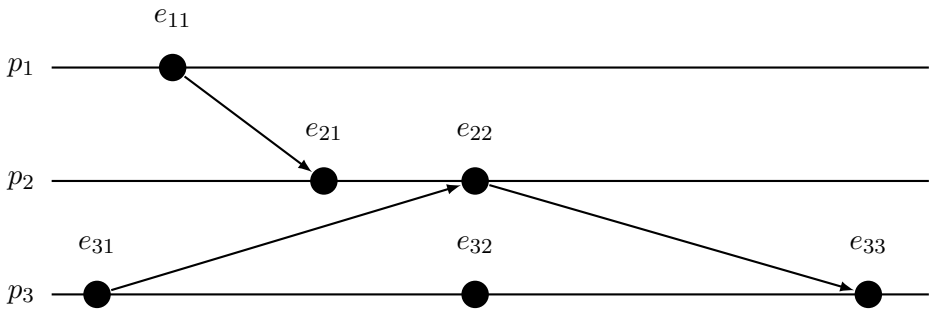



Figure 3.1: Process Event Diagram

### 3.2.2 Consistent and Inconsistent Cuts

For the remainder of this chapter, we denote the set of events in a system by $E$. Additionally, we assume the relation $\rightarrow_l$ defines the *local event order* of two events $e$ and $e'$, i.e., for two events $e$ and $e'$, $e \rightarrow_l e' \Rightarrow e \rightarrow e'$, and $e$ and $e'$ refer to the same process. We can then cut the set $E$ into two partitions, *past* and *future*, that comprise the events occurring before and after the cut, respectively. Formally:

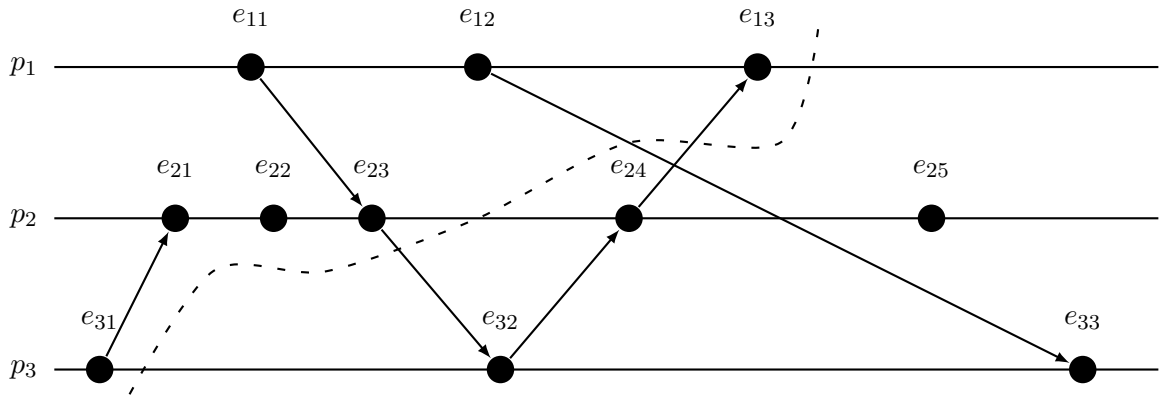
**Definition 1.** *A cut $C$ of an event set $E$ is a finite subset $C \subseteq E$ such that for two local events $e$ and $e'$, $e \in C$ and $e' \rightarrow_l e \Rightarrow e' \in C$ (see Mattern, 1989, p. 123).*

An example for a cut is depicted in Figure 3.2a. The dashed cut line is in compliance with our definition, because the local ordering of the individual events is taken into consideration, and a local event $e'$ that occurs before a local event $e$ is included in the *past* partition shown on the left side of the cut line. As can also be seen in the diagram, the event $e_{13}$ is triggered when process $p_1$ receives a message from process $p_2$ and is part of the cut. However, the corresponding send event $e_{24}$ is not. As Mattern (1989, p. 123) points out, "[s]uch a situation is undesirable because cuts are used to compute the global state of a distributed system along a cut line". This motivates the definition of a system with *consistent cuts* where every message that is received was also sent.

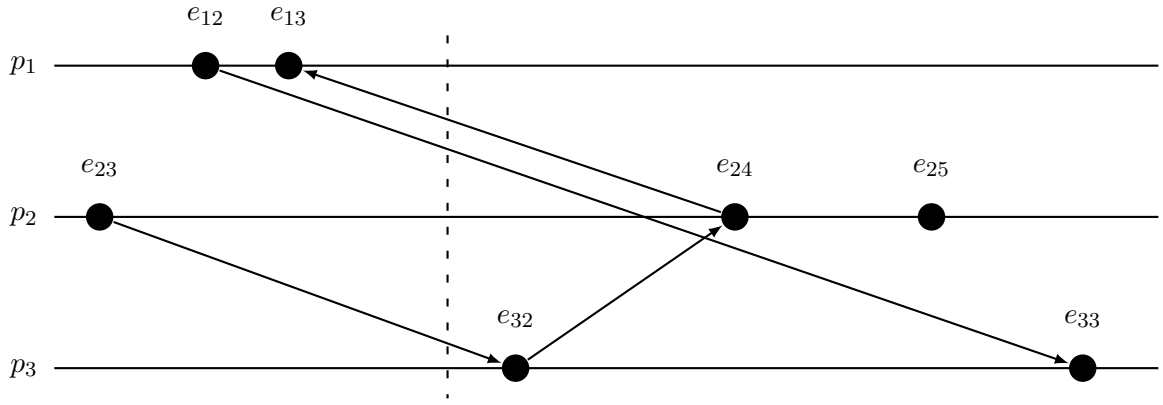
**Definition 2.** *A consistent cut $C$ of an event set $E$ is a finite subset $C \subseteq E$ such that for two events $e$ and $e'$, $e \in C$ and $e' \rightarrow e \Rightarrow e' \in C$ (Mattern, 1989, p. 123).*

To check whether a cut is consistent or not, Mattern (1989) proposes a simple "rubber band consistency test": Assuming a time line behaves like an idealized rubber band, it can be stretched and compressed until the cut is vertically fully aligned. If the message flow does not go backwards in time, i.e., there are no message arrows pointing from the right to the left, the cut is consistent, otherwise it is not.

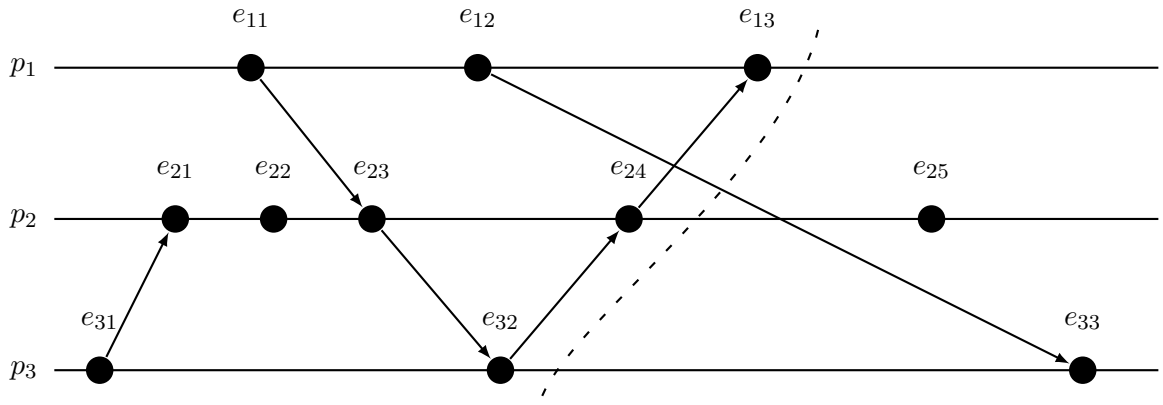
The aligned rubber band transformation for the previously described space-time diagram is illustrated in Figure 3.2b. It is clearly visible that the message flow between the two events $e_{13}$ and $e_{24}$ is reversed. Thus, the consistency test fails. By adapting the cut




(a) Example of an Inconsistent Cut




(b) Testing the Consistency of a Cut




(c) Adaption of the Cut

Figure 3.2: Consistent and Inconsistent Cuts

line as presented in Figure 3.2c, a valid and consistent set of past and future events can be created. Obviously, the cut shown in Figure 3.2c can also be vertically aligned without affecting the message flow. In general, Mattern (1989, p. 124) argues that "[f]or any time diagram with a consistent cut consisting of cut-events $c_1 \dots c_n$, there is an equivalent time diagram where $c_1 \dots c_n$ occur simultaneously, i.e., where the cut line forms a straight vertical line". The definitions outlined in this section will help us form the basis for our memory acquisition model. In the following, we explain the intricacies when generating a forensic image of a computer's RAM in more detail.

## 3.3 An Evaluation Model for Forensic Images of Physical Memory

Modern computers typically comprise two or more central processing units (CPUs) that permit the execution of multiple tasks in parallel. As such, a computer can be viewed as a miniature distributed system with concurrent system activities.

### 3.3.1 Events and Causality

Memory is a shared resource and too large to be accessed in its entirety in one atomic machine instruction. We therefore model every memory region that can be read or written with one atomic machine instruction as an individual *process* in terms of the terminology of distributed systems. An *event* is a read or write operation on such a region, performed by a program running on one of the available CPUs. The notion of potential causality is encoded in the programs that run on the CPUs. As an example, consider the two programs running on two processing units as illustrated in Figure 3.3. We denote the different memory regions that are accessed by the individual programs by $r_n$. A `read` instruction indicates that the value of a specific addressable memory region is read out. In contrast, a `write` instruction indicates that the value of a specific addressable memory region is updated and modified. As can be seen, the first instruction of the program running on core 1 *causally precedes* the second instruction which, in turn, causally precedes the third and forth operation. One possibility of a corresponding space-time diagram that visualizes the individual program steps and their causal dependencies is shown in Figure 3.4a. Another possible execution is depicted in Figure 3.4b. The only difference between the two executions is that the `read` operation invoked by core 2 on the memory region $r_2$ now occurs *after* the `write` operation of core 1 has been carried out.

Apart from the causal relation that is implicitly defined by the control flow of programs, there may be inter-program dependencies due to synchronization primitives that are implemented by the operating system, e.g., mutexes, locks, or messages. For instance, there exists a synchronization dependency between the two sample programs shown in Figure 3.5: Core 2 uses a mutex to deblock the program on core 1. Therefore, there is an explicit synchronization resulting in the fact that the program running on core 2 reads the value of the memory region $r_2$ before the program running on core 1 updates

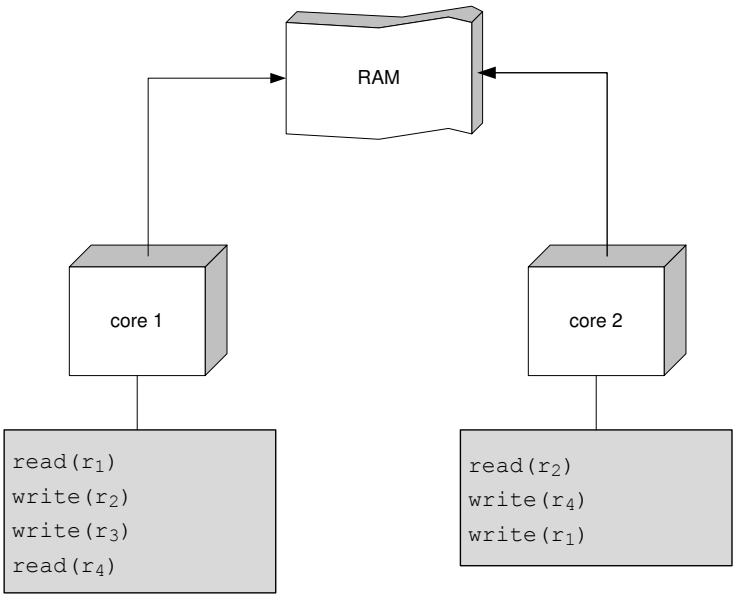



Figure 3.3: Sample System with Two Processing Units

said value by performing a write operation. Such dependencies must also be reflected in the causal relation and in the corresponding space-time diagram of the computation, respectively.

Based on the terms we have derived, we are now able to define the characteristics of a memory snapshot more clearly as well as explain major factors that may be used to assess its "goodness".

### 3.3.2 Memory Snapshots

We denote the set of all addressable memory regions by $\mathcal{R}$, the set of all possible values of a given memory region by $\mathcal{V}$, and the set of all timestamps by $\mathcal{T}$. To formalize a snapshot we need to refer to the contents of main memory at specific points in time. We therefore define the function

$$m : \mathcal{R} \times \mathcal{T} \to \mathcal{V}$$

that takes a specific memory region as well as a specific point in time and returns the contents of that memory region. So, for example, $m(x, y)$ refers to the value of memory region $x$ at time $y$.

**Definition 3.** *A* memory snapshot *(or simply* snapshot*) is a vector of tuples that, for every memory region, contains the value of that region together with the point in time the value was retrieved from this region.*

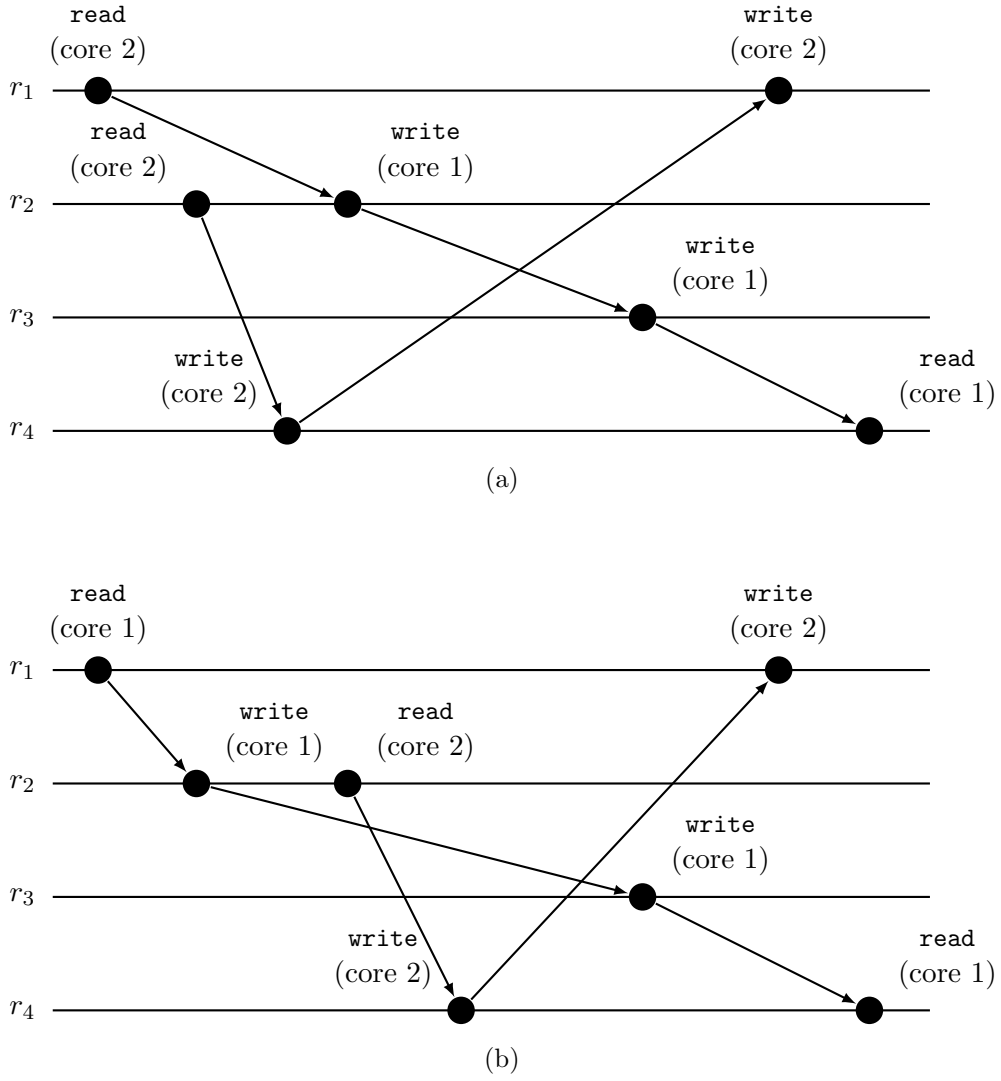



Figure 3.4: Possible Space-Time Diagrams for Two Programs

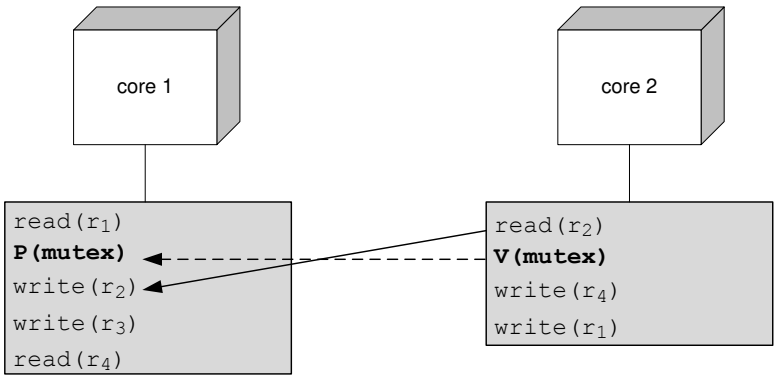



Figure 3.5: Use of Mutexes to Synchronize Program States

A snapshot is formalized by the function

$$s : \mathcal{R} \to \mathcal{V} \times \mathcal{T}$$

from memory regions to tuples $(x, y)$. For any such tuple we denote by $s(r).v$ the first component and by $s(r).t$ the second one. For example, if $s(r) = (x, y)$, then $s(r).v = x$ and $s(r).t = y$.

**Observation 1.** *Every snapshot corresponds to a cut through the space-time diagram of the system.*

A *full snapshot* covers *all* memory regions of the system, i.e., it stores a value for every memory region in $\mathcal{R}$. Sometimes, however, also *partial snapshots* are useful that only cover subsets $R \subset \mathcal{R}$ of all memory regions. Such snapshots are of interest when an investigator only wants to image specific parts of main memory, e.g., the address space of a particular program or the kernel address space of the operating system. The evaluation criteria we will describe in the following sections apply to both variants of memory snapshots. With respect to a partial snapshot, it is necessary to adapt the original "happened before" relation ($\to$) so that the principle of causality is also maintained in the reduced set of memory regions.

Let $\hat{E}$ be the set of events occurring in the set of memory regions $R \subset \mathcal{R}$. We define *causality with respect to a set of memory regions* $R \subset \mathcal{R}$, denoted by $\xrightarrow{R}$, as follows:

$$\xrightarrow{R} = \{(a, b) |\, a \in \hat{E} \wedge b \in \hat{E} \wedge a \to b\}$$

We can then compress the original space-time diagram for the entire memory address space $\mathcal{R}$ into a smaller *partial space-time diagram* that only comprises the subset of

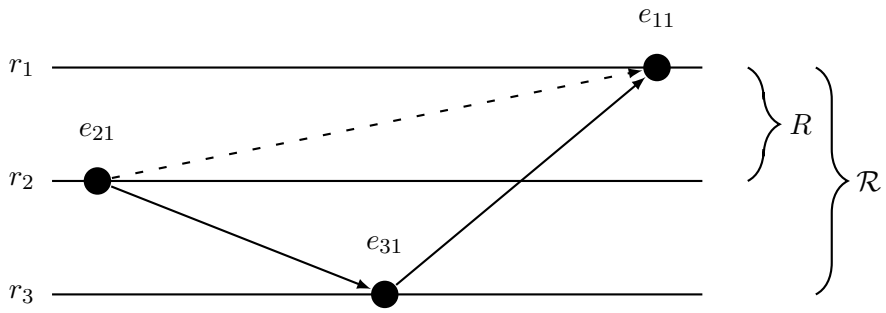



Figure 3.6: Construction of a Partial Space-Time Diagram

memory regions $R \subset \mathcal{R}$, but still maintains causality across all events involved. A transformation of a space-time diagram into a smaller version is depicted in Figure 3.6. In the example, the addressable space of main memory consists of only three memory regions for reasons of simplicity, i.e., $\mathcal{R} = \{r_1, r_2, r_3\}$. We assume that an investigator is interested in acquiring two of the three memory regions $R = \{r_1, r_2\}$. However, the causality relations that are defined in the original snapshot must also be reflected in its partial version. Therefore, as $e_{21} \rightarrow e_{31} \rightarrow e_{11}$ and, consequently $e_{21} \rightarrow e_{11}$ due to the transitivity condition, $e_{21}$ also *causally affects* $e_{11}$ *with respect to* $R$, i.e., $e_{21} \xrightarrow{R} e_{11}$ (as indicated by the dashed arrow line in Figure 3.6). We can generalize this conclusion and make the following observation:

**Observation 2.** *For two events $a$, $b$, $\{a, b\} \in \hat{E}$, the following condition holds:*

$$a \rightarrow b \Rightarrow a \xrightarrow{R} b$$

Please note that, in practice, a memory snapshot, no matter whether it is a full or partial variant, generally contains only raw physical data. The point of time when the value of an individual memory region is read out is usually not stored due to efficiency reasons. To define notions of correctness and integrity, however, we need to refer to this particular timestamp. It is therefore also vital to the definition of a memory snapshot.

### 3.3.3 Correctness of a Snapshot

Intuitively, correctness means that the snapshot contains only "true" values, i.e., those values that were actually stored in memory when the snapshot was taken. Correctness applies to any memory region contained in the snapshot, i.e., the concept can be applied to both full and partial snapshots.

**Definition 4.** *A snapshot is* correct with respect to a set of memory regions $R \subseteq \mathcal{R}$ *if for all these regions, the value that is captured in the snapshot matches the value that is*

*stored in this region at this specific point of time. Formally, a snapshot s is correct if it satisfies the following condition:*

$$\forall r \in R : s(r).v = m(r, s(r).t)$$

To explain the characteristics of a correct snapshot more clearly, consider the two sample algorithms depicted in Listing 3.1. For reasons of simplicity, we model regions of main memory to be imaged as an array `m` of size `n`, where `n` refers to the largest element of $\mathcal{R}$. The snapshot is stored in an array `s` of the same size. The first lines of the listing show an algorithm that produces a correct full memory snapshot. It iterates through main memory and copies the values of those regions to the snapshot array. On the other hand, the bottom of Listing 3.1 depicts an algorithm producing an *incorrect* snapshot as it only obtains the "true" values of every *second* memory region, i.e., regions ending with even numbers. We will see in Chapter 4 that such errors in the program logic – even though they are often more complex and, in contrast to this example, usually not intentionally inserted in the source code – may exist in memory acquisition solutions available on the market to date. It is still important to keep in mind though that the second algorithm will produce a correct *partial* snapshot if $R$ is restricted to the set of memory regions with even numbers.

Satisfying correctness may seem trivial at first glance. However, creating a correct copy of a host's volatile storage becomes significantly more difficult in the light of a possibly subverted operating system. For example, an early memory acquisition technique for Linux systems that makes use of a loadable kernel module (LKM) has been presented by Ring and Cole (2004). The technique relies on functions provided by the analyzed host in question though. Therefore, as the authors acknowledge, it is based on the questionable assumption that "[a] rootkit does not purposefully alter the behavior of how non-intruder related programs and files interface with the operating system" (Ring and Cole, 2004, pp. 165-166). By the same reasoning, we can argue that other (software-based) acquisition procedures suffer from similar limitations. We will describe the problem of correctly imaging memory of compromised systems in more detail in Section 3.6.

```
1  # Correct Memory Acquisition Approach
2  for i = 1 to n {
3      s[i] = m[i]
4  }
5
6  # Incorrect Memory Acquisition Approach
7  for i = 1 to n {
8      if i mod 2 == 0
9          s[i] = m[i]
10     else
11         s[i] = 0
12 }
```

Listing 3.1: Example of a Correct and Incorrect Memory Acquisition Algorithm

### 3.3.4 Atomicity of a Snapshot

An *atomic* snapshot should not show any signs of concurrent system activity. We formalize this by reverting to the theory of concurrent systems as outlined in Section 3.2.

**Definition 5.** *A snapshot is* atomic with respect to $\mathcal{R}$ *if the corresponding cut is consistent. Likewise, a snapshot is* atomic with respect to a subset of memory regions $R \subset \mathcal{R}$ *if the corresponding cut through the partial space-time diagram is consistent.*

With regard to the two examples presented in the previous section, we can see that the algorithms do not take any precautions to cope with concurrent system activity and, thus, will generally produce *non-atomic* snapshots. In the course of the respective imaging operations, it is, therefore, possible that concurrently running processes may access and change memory regions that have already been read out by the acquisition product. Libster and Kornblum (2008, p. 14) argue that these modifications cause "the captured data to be inconsistent" and lead to a "fuzzy snapshot" of the system state. In Listing 3.2 we have modified the two examples, using a special `lock` synchronization primitive. If `lock` is issued, all activities on the target system that are not related to the memory acquisition process are halted (i.e., frozen). Thereby, the atomicity of the respective operations can be ensured. Note that the second algorithm, although now being atomic, is still incorrect. This shows that correctness and atomicity are two independent properties of snapshots.

```
1  # Atomic and Correct Memory Acquisition Approach
2  lock
3  for i = 1 to n {
4      s[i] = m[i]
5  }
6  unlock
7
8  # Atomic, but Incorrect Memory Acquisition Approach
9  lock
10 for i = 1 to n {
11     if i mod 2 == 0
12         s[i] = m[i]
13     else
14         s[i] = 0
15 }
16 unlock
```

Listing 3.2: Example of an Atomically-Correct and Atomically-Incorrect Acquisition Algorithm

### 3.3.5 Integrity of a Snapshot

Even atomic snapshots are not taken instantaneously but require a certain time period to complete. The placing of this time period within a digital investigation is vital to the quality of a snapshot. For example, memory snapshots should be taken as early as possible within a digital investigation and not at the end when, e.g., activities of live analysis have permeated the system. The third property of memory snapshots, which we call *integrity*, refers to this aspect. Intuitively, integrity ties a snapshot to a specific point of time chosen by the investigator.

**Definition 6.** *Let $R \subseteq \mathcal{R}$ be a set of memory regions and $\tau \in \mathcal{T}$ be a point in time. A snapshot $s$ satisfies* integrity with respect to $R$ and $\tau$ *if the values of the respective memory regions that are retrieved and written out by an acquisition algorithm have not been modified after $\tau$. Formally:*

$$\forall r \in R : \tau \leq s(r).t \Rightarrow \exists t \in \mathcal{T} : \\ t \leq \tau \wedge \forall t' \in \mathcal{T} : t \leq t' \leq s(r).t : s(r).v = m(r, t')$$

In a certain sense, integrity refers to the "stability" of a memory region's value over a certain time period. In contrast, the correctness of an imaged memory region only refers to the "true" value of this region at a specific point of time. To illustrate the definition more clearly, consider the example space-time diagram in Figure 3.7. The example consists of four memory regions only, i.e., $R = \{r_1, r_2, r_3, r_4\}$. We assume that at time $\tau$, the imaging operation is initiated and leads to a change in the memory regions $r_3$ and $r_4$, as indicated by the dark-grey dots. We can, for instance, imagine that a software-based imaging solution is loaded into memory, thereby overwriting said regions. We can then find a specific time $t \leq \tau$ that represents the *original* state of memory, i.e., the state of memory that is solely defined by intrinsic system as well as process operations and is not yet affected by the imaging process. By $s(r_i).t$, $r \in \{1, 2, 3, 4\}$, we denote the point of time (as indicated by a black square) when the respective memory region is read out by the acquisition algorithm, and its value is saved to the snapshot.

Taking Definition 6 into account, the snapshot satisfies integrity with respect to time $\tau$ and memory regions $r_1$ and $r_2$. On the other hand, for memory regions $r_3$ and $r_4$, we can find point of times $t' \in \mathcal{T}$, $t \leq t' \leq s(r_i).t$, $i \in \{3, 4\}$, where the value that is stored in the snapshot does not equal the value that was originally stored in the respective memory region before $\tau$, i.e., a manipulation of said memory region has occurred in the course of the observation period. By $\tau$, we refer to the point of time when an investigator "decides" to take an image of a computer's memory. Although being highly subjective, this point of time ideally defines the very last cohesive system state before being affected (in any way whatsoever) by the imaging operation. As we have already indicated, $\tau$ should therefore mark a time very early in the investigation process. A first-class memory snapshot satisfies integrity with respect to this time and a preferably high number of memory regions.

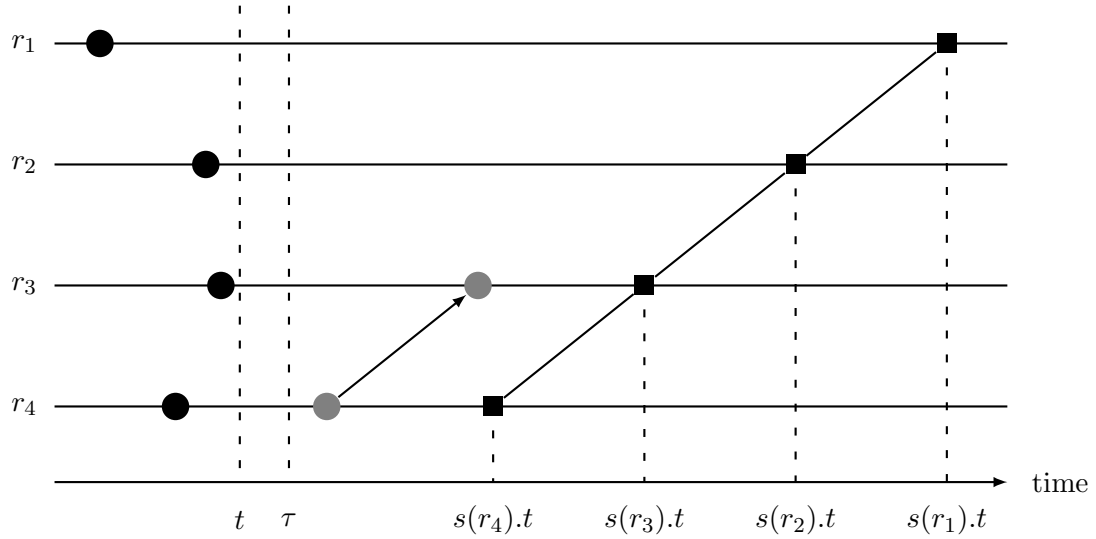



Figure 3.7: Assessing the Integrity of a Snapshot
with Respect to a Specific Point of Time $\tau$

Depending on the chosen $\tau$, the level of integrity of a generated snapshot can vary dramatically. Five sample benchmarks ($\tau_1$ to $\tau_5$) concerning two memory regions $R = \{r_1, r_2\}$ are shown in Figure 3.8. Similarly to Figure 3.7, a black dot indicates an event that is caused by system process operations, while a grey dot indicates the point of time the imaging operation is initialized, and an acquisition program is loaded into memory. By $s_1$, we denote the point of time when the first memory region is read out and saved to the snapshot. We assume that at time $s_2$, another, second snapshot is taken. It can be easily seen that the first snapshot does not satisfy integrity with respect to $R$ and times $\tau_1$ and $\tau_2$. It *does* satisfy integrity with respect to $\tau_3$ though. The second snapshot, on the other hand, satisfies integrity with respect to $R$ and all times past $\tau_3$. In particular, it satisfies integrity with respect to $R$ and $\tau_3$, even though the imaging program is already loaded into memory at that time. In dependance of the chosen $\tau$ (i.e., point of times very early in the investigation process versus later point of times where certain investigative measures have already been taken), the level of contamination of the target system due to a memory acquisition procedure can therefore be estimated quite accurately or only to a lesser degree.

It is important to note that a snapshot that satisfies integrity with respect to a set of memory regions $R \subseteq \mathcal{R}$ and a time $\tau \leq s(r).t$ for all $r \in R$ implies that the snapshot is both correct with respect to $R$ and atomic. On the other hand, satisfying correctness and atomicity does not automatically imply the integrity of the snapshot with respect to time $\tau$ as well: Considering the memory regions $r_3$ and $r_4$ from the example shown in Figure 3.7, we can construct a consistent cut through the partial space-time diagram and a partial snapshot that is correct with respect to $R = \{r_3, r_4\}$, although the original values in these memory regions have been overwritten.

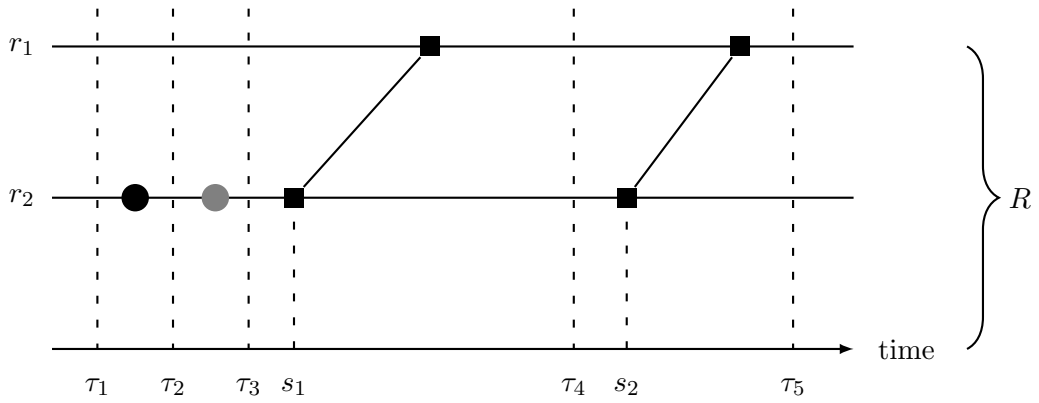



Figure 3.8: Assessing the Integrity of a Snapshot with Respect to Different Point of Times

Based on these observations, we can derive the following lemmas:

**Lemma 1.** *Let $s$ be a memory snapshot that satisfies integrity with respect to a set of memory regions $R \subseteq \mathcal{R}$ and a point of time $\tau$. If for all $r \in R$ holds $\tau \leq s(r).t$, then the snapshot is both correct and atomic with respect to $R$.*

**Lemma 2.** *The implication of Lemma 1 cannot be strengthened to an equivalence.*

## 3.4 Discussion of Forensic Soundness

In the previous sections we have shown that the three criteria we have identified, *correctness*, *atomicity*, and *integrity*, are independent and cannot be reduced to each other. On this basis, we postulate that all three criteria have to be satisfied for creating a forensic copy of physical memory. This leads to the following rule:

**Rule.** *The quality of a forensic snapshot is determined by its (degree of) correctness, atomicity, and integrity.*

So far, we have only broadly described when a snapshot should be considered "good" or "reliable", even though we have extensively discussed the respective influencing factors in the preceding sections. In fact, there has been an intense debate within the forensic community over the last years upon the meaning of and requirements for *forensic soundness*.[1] Classic definitions that mostly apply to the area of persistent media-oriented *dead forensics* stress, for instance, that a forensically-sound duplicate "must contain a copy of every bit" of the source in question and must not alter, "in any way", the respective data (Ball, 2005, p. 43). As various authors point out, these principles are often more of an ideal though and are especially hard to meet in the field of *live forensics*, e.g., when

[1] For a complete overview of the debate, please see the respective Yahoo Group at `http://tech.groups.yahoo.com/group/forensically_sound/`.

obtaining a forensic image of a computer's RAM (e.g., see Murr, 2006; Bejtlich, 2006). To resolve this conflict, Casey (2007, p. 49) suggests "looking at methods in traditional forensic disciplines, such as DNA analysis". Casey argues that these methods are regarded as "forensically sound", even though a DNA sample frequently smeared pieces of the original evidence. For this reason, he concludes that "[f]ocusing on whether an item of digital evidence was altered in any manner, rather than in a manner that affects the reliability or authenticity of the results, distracts from the substantive aspects of the evidence", and "imposing a paradigm of 'preserve everything but change nothing' is impractical and doing so can create undue doubt in the results of a digital evidence analysis, with questions that have no relation to the merits of the conclusions" (Casey, 2007, pp. 49-50).

We have attempted to consider the relevant aspects of this discussion by defining the correctness, atomicity, and integrity of a snapshot with respect to a set of memory regions $R \subseteq \mathcal{R}$. Thereby, a memory image may be regarded as "forensically sound" although parts of the image do not, for instance, reflect the "true" state of memory when the image was taken. Please note that, according to our definitions, we neither specify how big (or small) $R$ should be, nor what areas of memory $R$ should comprise. These aspects are intentionally left to be defined by the investigator (respectively by the the forensic community or vendors of memory acquisition solutions), i.e., an investigator ultimately decides the degree of forensic (un)soundness she expects and is willing to accept. A good acquisition method, of course, will seek to satisfy the different evaluation factors with respect to a set of memory regions $R$ that maximizes both information quantity (i.e., $R$ better approximates the set of all addressable memory regions $\mathcal{R}$) and quality (i.e., $R$ comprises areas of memory that contain information that are "essential" to the investigation). It is imaginable that vendors of memory acquisition solutions evaluate their products upon the performance they achieve for a given $R$. Thus, the difficult question of choosing a "forensically sound" acquisition method may, in the end, be reduced to simply comparing performance results for available technologies and choosing a solution with an "$R$ index" that best fits the needs and expectations of the investigator. In Chapter 4, we will assess the soundness of a number of common imaging solutions available on the market to date.

## 3.5 Integration of Existing Concepts into the Model

Before we outline typical challenges and limitations memory imaging approaches are frequently confronted with, we briefly illustrate how existing concepts established by Schatz (2007a,b) and Inoue et al. (2011b) can be mapped and integrated into our model.

At first glance, Schatz's (2007a, p. S128) notion of *fidelity*, i.e., "the copy of the memory (or the memory image) is a precise copy [of] the original host's memory", appears to correspond to our definition of *correctness*. However, Schatz proposes to measure the fidelity of a snapshot based on the level of atomicity an acquisition method can satisfy. As we have shown in the previous sections though, the correctness and atomicity of a snap-

shot are two orthogonal dimensions and, therefore, should be measured independently. Moreover, while Schatz only vaguely describes the different criteria by outlining various factors that may, for instance, affect the atomicity of a snapshot, our definitions are more specific and actually permit to quantify the degree of correctness, atomicity, and integrity a memory image achieves. Schatz (2007a, p. S128) also emphasizes determining the *reliability* of an acquisition method, i.e., to check whether the method produces a "trustworthy result or none at all" and may be compromised by malicious activity. This is an important requirement, especially with regard to a later forensic analysis, and is reflected in our model by assessing the *correctness* of a snapshot (which stipulates that the state of memory that is saved at a specific point of time represents the "true" state of the memory in question). The last factor that Schatz considers to be notable is the *availability* of a given memory imaging solution, i.e., the solution must be "working on arbitrary computers (or devices)" (Schatz, 2007a, p. S128). This factor does not affect the quality of a snapshot in the first place and, therefore, has been neglected in our discussion. However, as we have pointed out in Chapter 2, the availability of a technique is nonetheless highly relevant in practice and, thus, must be taken into account when trying to classify memory acquisition approaches.

In contrast to the work of Schatz, Inoue et al. (2011b) have introduced four evaluation metrics for physical memory snapshots: *Correctness*, *completeness*, *speed*, and the *amount of interference*. The perception of *correctness* that Inoue et al. (p. S43) have, i.e., "the physical address of a page in the image [corresponds to] the actual physical address of that page in memory", is not as strict as the one we have proposed in this thesis and does not explicitly take the actual data into consideration that is stored in said page. With regard to the notion of *completeness*, i.e., "all of the physical address space which is not allocated to devices or the BIOS", we can easily form a corresponding set of memory regions $R$ that comprises the address space in question. On the other hand, our definition is more flexible and permits generating partial snapshots that are "complete" concerning specific parts of memory, e.g., the kernel address space or the address space of an application.

The metric of *speed*, although of high importance in practice, should not be regarded as a primary evaluation criterion for a memory acquisition technique in our opinion. Speed is desirable but does not directly influence the forensic soundness of an image. As long as an approach is capable of producing a snapshot of a computer's RAM that satisfies correctness, atomicity, and integrity to a degree that is acceptable for the investigator, the performance of the respective operation should only be considered in the second place. Last but not least, the authors' definition of *amount of interference*, i.e., "the amount of memory [an acquisition program] alters on the machine", roughly corresponds to our level of integrity. By monitoring memory regions that are updated when the memory imaging operation is initialized, we cannot only estimate the *impact* of the respective solution (thereby "discriminating against" software-based approaches), but we are able to determine the level of concurrent system activity that "smears" the snapshot as well. In contrast to prior works, we have also formalized the criteria for evaluating the "soundness" of a memory image. With the help of our definitions, it is

possible to derive suitable metrics for measuring the quality of a forensic snapshot and, thus, help vendors of forensic acquisition solutions assess and drive the development of their products forward.

## 3.6 Critical Perception of Current Technologies

In addition to our explanations given for different memory acquisition approaches in Chapter 2, we now briefly describe how current technologies are capable of roughly satisfying the criteria of correctness, atomicity, and integrity introduced in this chapter.

As we have argued in Section 3.3.3, creating a correct snapshot of a computer's RAM is particularly demanding in the light of possibly installed malicious applications. For example, a rootkit may attempt to prevent the imaging operation directly by blocking access to certain system structures, e.g., to the internal `\\.\Device\PhysicalMemory` section object (see Section 2.2.5). Kornblum (2006) has pointed out though that such manipulations do not reflect normal system behavior and, thus, easily indicate the presence of a security threat. Malicious programs may, however, also intervene with the acquisition process more subtly. For example, Bilby (2006) has demonstrated how memory imaging may be subverted by hooking the System Service Dispatch Table (SSDT), a core system table that contains pointers to system service routines. Likewise, Sparks and Butler (2005) have presented a prototype that is able to dissemble different views of virtual memory to applications by manipulating the Translation Lookaside Buffer (TLB), a CPU cache that stores recently used memory pages for accelerating the virtual-to-physical address translation process. Last but not least, as we have already discussed in Chapter 2 as well, even hardware-based approaches that are, for instance, invoked over the FireWire interface and are independent from functions of the target operating system are prone to attack and may lead to false results. As can be seen, even though the correctness of a memory image is crucial for establishing a "ground truth" with regard to a later investigation, it is non-trivial to achieve in practice when the integrity of the target system is unknown. Even if the trustworthiness of a computer can be reasonably assumed, (unintentional) errors in the program code of an imaging solution may distort the correctness of a created memory snapshot and impede a later analysis of the captured data. We will see examples for such scenarios in Chapter 4 of this thesis.

Ensuring the *atomicity* of a snapshot may be significantly difficult in practice as well. It is to be expected that software solutions such as *mdd* (ManTech CSI, Inc., 2009), *Memoryze* (Mandiant, 2011), or *MoonSols* (Suiche, 2013) do not sufficiently prevent concurrent system activity and, therefore, are incapable of creating an atomic image of a computer's RAM. By the same reasoning, we can assume that these techniques significantly affect the level of *integrity* of produced memory snapshots. In particular, we assume that applications with a complex graphical user interface such as *FTK Imager* (AccessData, 2012) lead to a higher degree of memory contamination than more lightweighted products that do not require any additional libraries. Similar concerns also apply for applications used on other platforms. For instance, with regard to *Linux*

operating systems, the *memdump* utility may be invoked to retrieve contents of physical memory over the `/dev/mem` device file (Farmer and Venema, 2005a,b).[2] In the course of this operation, other processes continue running though and, thus, may change the respective data. The latter issue is partially addressed by Ring and Cole (2004). The authors implement a loadable kernel module that is capable of blocking rescheduling of other processes by setting a special "zombie flag" in the corresponding task structure. Thereby, the level of concurrent activity may be reduced, and the atomicity of the imaging procedure is increased.

Acquisition approaches that are based on dedicated hardware cards or that operate on the firmware level (see Sections 2.2.1 and 2.2.3) promise true atomicity as they freeze the system state before starting to copy memory. The same is true for virtual machines that permit suspending the execution environment. All these approaches require special, preparatory measures though and are therefore only suited for specific scenarios. With respect to virtual machines, it is also important to point out, that the correctness and integrity of a memory image has not been examined in detail yet. For instance, a discussion on a memory forensics-related mailing list has indicated that volatile information are not completely reflected in the respective snapshot file when a virtual machine is temporarily paused, and the contents of memory is saved to hard disk (Volatile Systems, LLC, 2012).

Estimating the performance of alternative operating system injection methods as the one illustrated by Schatz (2007a) remains difficult at the time of this writing. On the one hand, such an approach offers a high level of atomicity, because the host operating system is stopped. On the other hand, the technique requires both the acquisition OS as well as auxiliary software to be loaded into RAM. The actual impact of these activities on the level of integrity still needs to be more closely evaluated, but first measurements by Schatz suggest it is notable. According to the author, the technique leads to a 35% difference in memory pages compared to a reference image. Further testing is required in the future, however, to confirm these results.

## 3.7 Summary

In this chapter, we have defined three factors for evaluating the quality of a forensic memory snapshot: *correctness*, *atomicity*, and *integrity*. A *correct* snapshot contains the actual values of (a set of) addressable memory regions that were stored in those regions at a specific point of time. In contrast, an *atomic* snapshot is free of the signs of concurrent system activity. By the level of *integrity*, we can estimate how "stable" values of memory regions remain in the course of the imaging operation. In particular, we can estimate the "impact" of a certain acquisition approach and "discriminate against"

[2] The `/dev/mem` device file is a special file that serves as a representation of physical memory. Please note that on more recent Linux systems, access to this object is frequently restricted with the help of a special kernel configuration option (see van de Ven, 2008).

more invasive, software-based methods that must be loaded into memory and, thereby, potentially destroy valuable pieces of evidence.

In order to produce a high-quality memory snapshot, acquisition techniques must satisfy the presented factors to a high degree. As we have illustrated, however, this is often a non-trivial task in practice. For instance, if the target operating system is subverted by a malicious application, the correctness of the captured data may be at stake, especially when an imaging solution internally relies on functions that are provided by said system. Likewise, if concurrent activity is not sufficiently prevented during the acquisition process, the level of atomicity may be significantly affected. Taking these aspects into consideration, it is important to assess forensic tools upon whether or not or to what degree they meet the criteria described above. With the help of the model presented in this chapter, investigators have a starting point for comparing available technologies more reasonably and choosing a product that best fits their needs and expectations. The architecture of an evaluation platform that permits measuring the performance of software-based solutions in greater detail is subject of the following chapter.

## Chapter 4

# An Evaluation Platform for Forensic Memory Acquisition Software

With the help of the criteria we have introduced in the previous part of this thesis, we are now able to determine the quality of a forensic memory snapshot more thoroughly and, thus, assess the performance of an acquisition solution in more detail. In this chapter, we present the architecture of an evaluation platform that helps quantify the degree of *correctness*, *atomicity*, and *integrity* of selected imaging applications in an in-depth and repeatable manner. For this purpose, we execute the software on top of a highly customized version of the *Bochs* x86 PC emulator (The Bochs Project, 2013a). The adapted emulator serves as the foundation for our platform and forms the starting point for the experiments and measurements we will outline in a later section. Our approach is based on a *white-box testing* methodology, i.e., we inspect the source code of an acquisition utility in question and slightly adjust it to our needs. By specifying a number of *hypercalls* that are inserted in the program code, we are able to keep track of important system events and operations, e.g., the point of time when the imaging process is initiated or finished, respectively, or when a page of RAM is about to be copied. Our platform intercepts these hypercalls, creates a protocol of the different activities, and generates its own view of the system state. This view is matched with the produced memory snapshot in a later analysis phase to derive the value of the individual performance factors. With our method we are able to answer specific questions concerning the functionality and operability of imaging applications. These include, for instance:

- Does an acquisition utility produce a snapshot that equals the size of the physical address space?
- Does the created snapshot contain the data that was stored in a memory page at the time said page was imaged?
- How does an acquisition utility cope with errors and areas of memory that cannot be accessed?
- In how far does concurrent activity interfere with the imaging process?
- What is the impact of an acquisition utility, and how much memory is changed when the application is loaded into RAM?
- What is the amount of memory that is changed in the course of the imaging period?

The approach we describe in this chapter is generally applicable to all acquisition applications for which access to source code is given. We exemplify this by focusing on

three popular imaging utilities for the family of Microsoft Windows operating systems, i.e., *win32dd* (Suiche, 2009b), *mdd* (ManTech CSI, Inc., 2009), and *WinPMEM* (Cohen, 2012b). As we will see, two of the three evaluated products initially produced forensic memory snapshots that differed both in size and contents. In more detail, the respective solutions ignored regions of the physical address space that were used by hardware devices. As a consequence, the offset mapping of subsequent memory areas was corrupted. Because such errors may lead to potential data misinterpretations in the course of the evidence investigation phase, they must be critically judged. We have developed patches for the affected program components so that all evaluation candidates eventually generated correct images of a computer's RAM. As we will also see, in dependence of the time that is required for the imaging process, maintaining the atomicity and integrity of a snapshot gets increasingly more difficult, even on idle systems. We will illustrate in a later section how the degree of such consistency violations can be estimated with the help of our platform.

**Outline of the Chapter**

The remainder of this chapter is outlined as follows: In Section 4.1, we give a brief overview of existing models and concepts for determining the quality of memory acquisition solutions. We also briefly illustrate major characteristics of the standard memory imaging process on Microsoft Windows operating systems. In Section 4.2, we describe our evaluation methodology as well as the architecture of our testing platform. A study of three software-based acquisition utilities and their corresponding performance results are subject of Section 4.3. In Section 4.4, we explain the advantages and disadvantages of our approach and discuss possible alternatives. In addition, we outline a number of weaknesses and limitations our platform still has to cope with at the time of this writing. We conclude with a short outlook on future research possibilities and a summary of our work in Sections 4.5 and 4.6.

## 4.1 Background Information

### 4.1.1 Existing Work

Existing literature on the quality or impact of imaging solutions is unfortunately still sparse, and the majority of researchers has solely focused on assessing a memory snapshot with respect to a single factor. A noteworthy exception is Schatz (2007a, p. S128), who sketches a general process for duplicating volatile information in a forensically sound manner. According to the author, an acquisition method is thus "ideal" if it produces an image of physical RAM that "is a precise copy [of] the original host's memory" and is "available, working on arbitrary computers (or devices), and additionally [is] reliable, either producing a trustworthy result or none at all".

Even though Schatz outlines the previous requirements in more detail in the remainder of his paper, his explanations rather aim at illustrating the limitations of current technologies than at finding accurate metrics for sound memory imaging. In contrast, Inoue et al. (2011b) suggest the four factors *correctness*, *completeness*, *speed*, and the *amount of interference* for determining the quality of forensic images (see Chapter 3). Using so-called graphical *dotplots*, they are able to reveal and visualize systematic errors in memory snapshots. Walters and Petroni (2007) attempt to estimate the degree of system contamination that is caused by acquisition-related activities. For this purpose, they create so-called snapshot "baselines" and match the system state before and after an utility has been launched. Similarly, Sutherland et al. (2008) pursue an empirical approach and observe the system state over the time a memory snapshot is generated. The main objective of their study is to assess common imaging programs by monitoring modifications of certain system resources, e.g., the Windows registry or the hard disk partition.

With respect to the latter works, Lempereur et al. (2010, p. 3) argue that "[w]hile [the authors'] results and method can serve as useful guidelines for the developers and users of forensic software, the selection of metrics [is] arbitrary and may not represent the best way to quantify the impact of forensic tools". Lempereur et al. therefore propose comparing "the state of a machine on which forensic acquisition has been performed, to the state of an identical, unaltered, machine". To illustrate the practicability of their approach, they set up several virtual machines and correlate the results of the individual test runs with the help of a Python-based framework.

Last but not least, Su and Wang (2011) present a statistical model for calculating the probability a memory area is changed when loading an imaging program into RAM. A similar methodology is pursued by Savoldi et al. (2010) in order to quantify the degree of "uncertainty" due to executing an imaging solution on a machine. These concepts are helpful in fostering a better understanding of the challenges in forensic memory acquisition, yet they only aid little in recommending a specific solution in practice.

In the following section, we will give a brief overview of the imaging process as it is frequently applied on Microsoft Windows operating systems.

### 4.1.2 Forensic Memory Imaging on Microsoft Windows Operating Systems

A common technique for acquiring a (raw) forensic copy of a computer's RAM relies on leveraging the internal `\\.\Device\PhysicalMemory` section object. As the name suggests, the object provides access to sections of physical memory. However, as we have explained in Section 2.2.5, permissions to open the resource in user space were revoked with the introduction of Microsoft Windows Server 2003 (Service Pack 1) for security reasons (Microsoft Corporation, 2013b). For this reason, all of the imaging applications we considered for our evaluation did not only consist of a user-mode administration program, but also of a kernel-level driver. The kernel driver is responsible for obtaining a handle to the section object. For this purpose, the `ZwOpenSection` function is usually called in

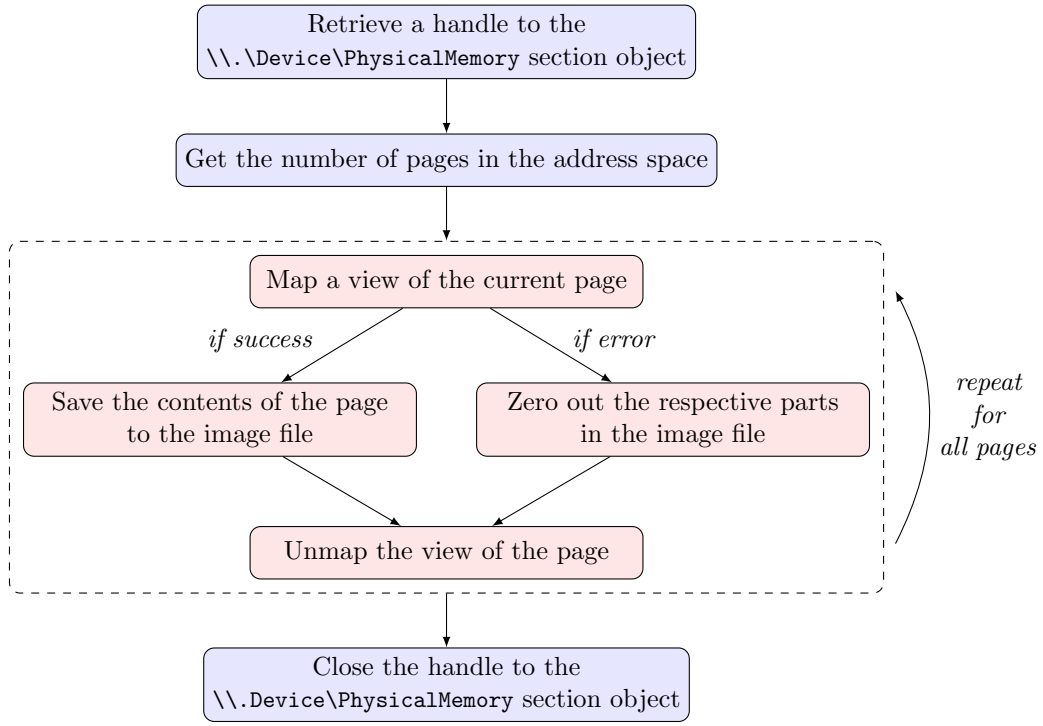



Figure 4.1: Sample Algorithm for Acquiring a Forensic Copy of Memory

the first step. After the size of available memory has been determined, portions of RAM may then be read out page wise, for instance, with the help of the `ZwMapViewOfSection` function. In the last step, a mapped section can either be directly written to the image file in kernel space or transferred to user space via a buffer for further processing.

An illustration of a typical imaging algorithm is depicted in Figure 4.1. As can be seen, the design of an acquisition solution can be quite simple in practice. Particularly error-prone program parts include determining the capacity of the physical address space though. In Section 4.3.2, we will discuss how miscalculations in this area may corrupt a memory snapshot. Other examples for nuisances in imaging applications are implicitly assuming the correctness of conducted operations and not – or not sufficiently – testing the return values of called functions. For instance, Stüttgen and Cohen (2013) point out that it is possible to induce a fatal error in the majority of imaging solutions available on the market to date by simply manipulating the `MmGetPhysicalMemoryRanges` function we will describe in more detail in a later part of this chapter. Likewise, Milković (2012) demonstrates various anti-forensic techniques for blocking or evading the imaging process. In this regard, it is also important to emphasize that the procedure of invoking the `\\.\Device\PhysicalMemory` object may also be susceptible to manipulation (see Bilby, 2006). Therefore, more sophisticated solutions frequently support alternative acquisition methods as well, e.g., via the `MmMapIOSpace` kernel routine (Microsoft Corporation, 2013e). In the scope of this thesis, the latter approaches have not been evaluated though.

## 4.2 Measurement Methodology and Platform Architecture

As mentioned in the beginning of this chapter, our evaluation is based on a *white-box testing* approach, i.e., we examine the source code of the desired acquisition software, identify the relevant instructions that are responsible for reading and writing out portions of memory to the respective image file, and slightly adapt the code to our needs by inserting a number of *hypercalls*. The different hypercalls are intercepted and processed by our platform in the course of the imaging period to create an external view of the system state. By matching this view with the produced memory snapshot, its level of *correctness*, *atomicity*, and *integrity* can later be determined. In the following sections, we describe the design, functionality, and mode of operation of our platform in more detail.

### 4.2.1 Platform Architecture

As we have already indicated, our testing platform is built upon *Bochs*, an open source x86 PC emulator (The Bochs Project, 2013a). Bochs' distinct advantages over similar products such as *QEMU* are its smaller code base by roughly 50% ($\sim$ 250,000 lines of code) and the possibility to implement the stub of a custom *instrumentation interface* comparatively easily. Our interface is largely written in C, apart from performance-critical parts that were developed in assembly language, and provides a series of callback functions that are invoked when specific system events and operations occur. In particular, with the help of the `bx_instr_lin_access` function, we are able to monitor linear memory accesses of the guest system. We will see in a later section of this paper that this capability is beneficial for determining the degree of atomicity.

As we have also pointed out already, the guest system can communicate with the emulator via a number of *hypercalls* that are triggered in the case of imaging-related activities. Technically, a hypercall uses the `EAX` register to indicate a specific event and pass additional meta information to the instrumentation interface if required. By issuing a *breakpoint* command (`INT 3`), the respective hypercall is executed in the next step. In Listing 4.1, a sample hypercall is depicted. In the example, the beginning and end of a page acquisition operation is signaled with the `HCALL_BEGIN_IMG_PAGE` and `HCALL_BEGIN_END_PAGE` constructs, respectively (Lines `39` and `47`). The instructions invoke the `EXECUTE_HYPERCALL_SYSTEM_MODE` function (Line `17`) that expects four parameters, including an offset to the imaged region in the physical address space. By truncating the lower 12 bits of the offset (Line `20`), the corresponding page number is determined.[1] This value, in combination with the page operation and an indicator whether the operation was successful or not (`PAGE_TYPE_ACCESSIBLE` or `PAGE_TYPE_INACCESSIBLE`), is finally saved in the `EAX` register before the breakpoint interrupt is triggered (Line `25`). We have patched Bochs such that the interrupt is intercepted and gets solely processed by our interface. Before control is returned to the guest operating system, the vector

[1] As explained in Section 2.1.2, the upper 20 bits of a 32-bit address define the physical page number.

**Platform Definition File <hypercall.h>**

```
enum page_types {
    PAGE_TYPE_ACCESSIBLE      =  0,
    PAGE_TYPE_INACCESSIBLE    =  1,
};

typedef union HCALLARGS_T {
    unsigned int raw;
    struct {
       unsigned int op        :  4;
       unsigned int page_type :  1;
       unsigned int page      : 20;
   };
} HCALLARGS;

#define EXECUTE_HYPERCALL_SYSTEM_MODE(ARGS, OP,             \
                                      VIEWBASE, PAGE_TYPE) \
do {ARGS.op = OP;                                          \
    ARGS.page = VIEWBASE.LowPart >> 12;                    \
    ARGS.page_type = PAGE_TYPE;                            \
    __asm {                                                \
    __asm lea ebx, ARGS                                    \
    __asm mov eax, [ebx]                                   \
    __asm int 3                                            \
    }                                                      \
} while (0)
```

**Kernel Driver File of the Memory Acquisition Software**

```
// reference the UNION
HCALLARGS args;

...

// inform the platform of the beginning of a page acquisition
// operation
EXECUTE_HYPERCALL_SYSTEM_MODE(args, HCALL_BEGIN_IMG_PAGE,
                              <Offset>, PAGE_TYPE_ACCESSIBLE);

// internal imaging operations
...

// inform the platform of the end of a (successful) page
// acquisition operation
EXECUTE_HYPERCALL_SYSTEM_MODE(args, HCALL_END_IMG_PAGE,
                              <Offset>, PAGE_TYPE_ACCESSIBLE);
```

Listing 4.1: Example of a Hypercall

| **Hypercall** | **Description** |
|---|---|
| HCALL_BEGIN_LOG | Signals the beginning of the memory acquisition process. |
| HCALL_END_LOG | Signals the end of the memory acquisition process. |
| HCALL_BEGIN_IMG_PAGE | Signals the beginning of an image operation to the platform for each page. |
| HCALL_END_IMG_PAGE | Signals the end of an image operation to the platform for each page. |
| HCALL_LOAD_ACQUISITION_SOFTWARE | Notifies the platform of the point of time shortly before the memory acquisition program is loaded into memory. |
| HCALL_END_ACQUISITION_SOFTWARE | Notifies the platform of the point of time the acquisition program has completed its operations and is unloaded from memory. |
| HCALL_QUIT_SIM | Notifies the platform that the imaging process has finished and that the guest should be shut down. |
| HCALL_LOAD_IMAGER_CONFIG | Notifies the platform that more information about a certain acquisition program is requested for a test run. |
| HCALL_SIGNAL_PAGE_SIZE | Notifies the platform of the page size. |
| HCALL_SIGNAL_NUMBER_OF_PAGES | Notifies the platform of the number of memory pages. |

Table 4.1: List of Hypercalls Processed by the Instrumentation Interface

is discarded. Thereby, the operation appears as completely transparent to the guest. Due to this behavior, applications running on the emulated machine can no longer be debugged. We believe that this loss of functionality is acceptable in the scope of our evaluation though. An overview and short description of the hypercalls we have defined is given in Table 4.1. As can be seen, several calls are reserved for administrative purposes, e.g., for externally shutting down and resetting the guest system or for signaling the size of the physical address space to Bochs (see also Section 4.3.2). On the other hand, most hypercalls are directly related to the imaging process. For instance, the HCALL_LOAD_ACQUISITION_SOFTWARE instruction is invoked by a small wrapper program in order to mark the point of time before the acquisition program is loaded into memory. We will illustrate in the following section how, based on the individual hypercalls, the different factors for sound memory imaging can be measured.

### 4.2.2 Measuring Factors for Sound Memory Imaging

#### Correctness

In order to determine the correctness of an acquisition solution, we create a view of the guest's physical address space as it is seen by Bochs, in parallel to the imaging process. This external view is then matched with the produced snapshot in a later analysis phase to identify possible deviations. Specifically, when an imager accesses a memory page for subsequent duplication (e.g., via the `ZwMapViewOfSection` function), the respective page number is signaled to the instrumentation interface. Our platform then creates a concomitant copy of the memory region the page is stored in before passing control back to the guest system. The algorithm for acquiring the contents of memory is shown in Listing 4.2. The depicted function `acquire_page` expects a pointer to the desired page in question, its offset in the physical address space as well as the page size, and is invoked each time the `HCALL_BEGIN_IMG_PAGE` hypercall is received.

On the first function call, the external snapshot file (`dump_file`) is set up. For this purpose, the `posix_fallocate` interface of the system is used (Line `12`) that ensures sufficient storage capacity for the parallel image is provided on the machine. After the (empty) snapshot has been successfully created, the file is mapped into the host's virtual memory (Lines `21-24`), and the page acquisition process is initiated in the following (Lines `38-46`). For performance reasons, data are thereby written out to the respective file offsets eight bytes at a time. Please note that while the latter operations are atomically executed, mapping a page inside the guest system and informing the host platform of this process are generally not, because we wanted to reduce modifications of the original imaging code to a minimum. Consequently, there is a small time frame in which a page may be updated, and the respective changes are reflected in the memory snapshot of the host system but not of the acquisition utility (or vice versa). As we will see in Section 4.3.2 though, these differences comprise only a few bytes and are therefore negligible in our opinion. In sum, with the help of the described method, we are able to verify whether the size of a forensic memory snapshot equals the size of the physical address space as well as whether the data stored in the image file corresponds to the contents of memory at the time the snapshot was taken.

#### Atomicity

Directly measuring the level of atomicity a forensic memory image satisfies turned out to be infeasible in our experiments. For this reason, we pursued an *indirect* approach and attempted to quantify the degree of *atomicity violations*. An atomicity violation occurs when a memory region that has already been imaged is accessed by a running thread, and, based on this access, another memory region is modified that still needs to be duplicated. In this case, the snapshot becomes inconsistent, because an effect is reflected in the image file without including the corresponding cause. For instance, in the space-time diagram depicted in Figure 4.2, a concurrent thread (as indicated by the

```
void acquire_page(unsigned long long *page,
                  unsigned long long page_offset,
                  size_t page_size) {

    // initialize the external physical memory snapshot file
    if(first_acquisition) {
        LOG("Creating dumpfile with size %s.",
            commaprint(pMemoryStructure->memory_size));

        // check whether sufficient storage capacity can be
        // allocated for the external snapshot file
        if (posix_fallocate(dump_file, 0,
            pMemoryStructure->memory_size)) {

            // the allocation failed
            (...)
        }

        // create a mapping of the external snapshot file
        // into virtual memory
        dump_mapping = (unsigned long long *)
            mmap(NULL, pMemoryStructure->memory_size,
            PROT_READ | PROT_WRITE, MAP_SHARED,
            dump_file, 0);

        // check whether the mapping operation succeeded
        if (dump_mapping == MAP_FAILED) {
           (...)
        }

        first_acquisition = false;
    }

    // perform further file sanity checks
    (...)

    // locate the offset in the snapshot file
    unsigned long long *mapped_page = dump_mapping +
             (page_offset / sizeof(unsigned long long));

    // write the desired page out to the snapshot file
    for(size_t position = 0; position < page_size /
        sizeof(unsigned long long); position++) {

        *(mapped_page + position) = *(page + position);
    }
}
```

Listing 4.2: Algorithm for Generating the External Physical Memory Snapshot

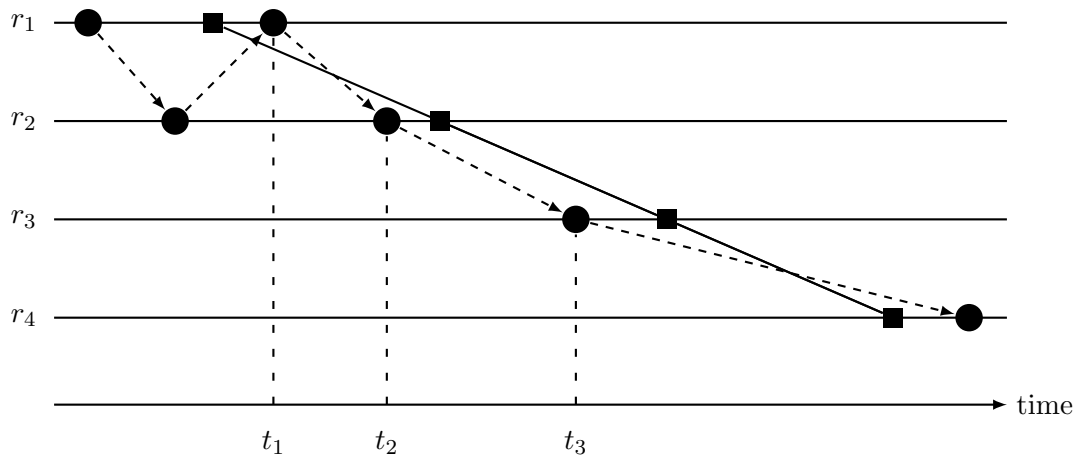



Figure 4.2: Example of an Atomicity Violation

black circles) accesses the memory regions $r_1, r_2, r_3, r_4$, while an acquisition program (as indicated by the black squares) is being executed. In the given example, the thread accesses the memory region $r_1$ at time $t_1$ *after* the region has been imaged, and, in the following, manipulates memory regions $r_2$ and $r_3$ at times $t_2$ and $t_3$, although these are not yet represented in the snapshot and will be written out at a later time. Consequently, with respect to the regions $r_2$ and $r_3$, the atomicity criterion is violated.[2]

In practice, it is hard to verify whether the memory operations of a thread are truly *causally related*, i.e., whether changing the value of one memory region (directly) leads to the change of another region (see Chapter 3.2). With respect to *all* threads running on a machine, we believe this task is almost impossible to carry out efficiently, because otherwise, we would have to dynamically taint track the entire system. Due to the previously described arguments, our platform measures a slightly weaker metric, namely the degree of *potential atomicity violation*, i.e., atomicity violations that are unspecified whether they effectively lead to a modification of the respective program flow. As such, the metric is an *upper bound* for the degree of atomicity violation. The complement of this value, on the other hand, represents the absolute and accurate *minimum* degree of atomicity a snapshot satisfies.

To quantify the amount of potential atomicity violations, we hook the callback function `bx_instr_lin_access` as indicated in Section 4.2.1, observe all memory operations once the imaging procedure has started, and keep track of the pages that have already been acquired. If a thread accesses a page after it has been imaged, it is inserted into a self-balancing binary search tree that serves as a *watchlist* for potential atomicity violators. The currently running thread is identified with one of two methods, depending on the processor privilege level (*ring 0* or *ring 3*) the corresponding process is executed on.

[2] Technically, the atomicity criterion is violated with respect to memory region $r_4$ as well. This violation is of no relevance in the example, however, because the memory region has already been acquired at the time the violation occurs.

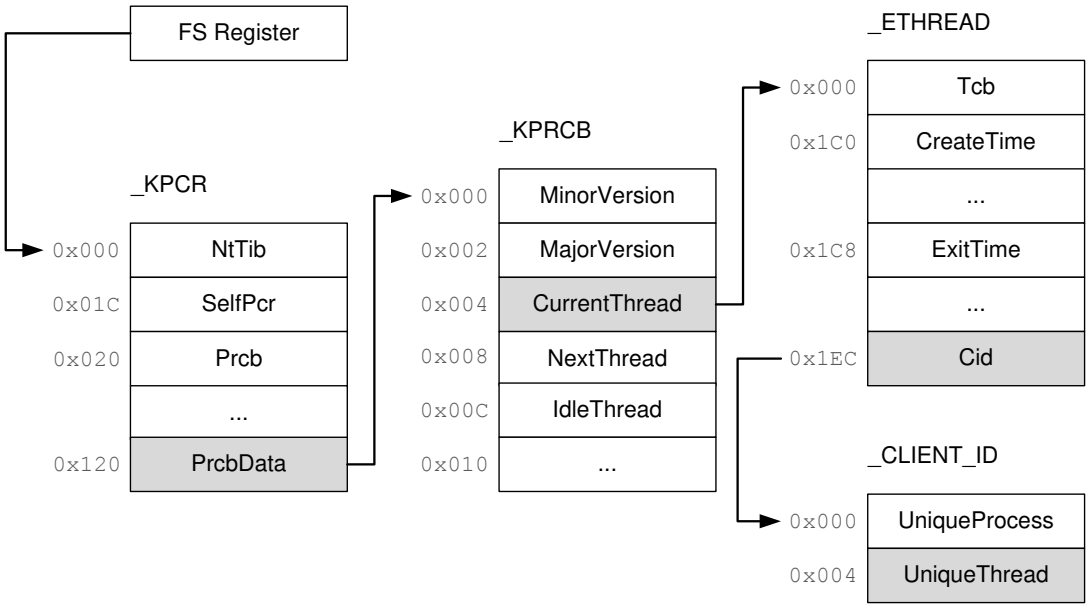



Figure 4.3: Retrieving the Identification Number of a Kernel-Level Thread in Privileged Mode

For user-level threads, we obtain a pointer to the *Thread Environment Block* (TEB) that is referenced at an offset of the `FS` segment register (Schreiber, 2001). The TEB, in turn, points to a `_CLIENT_ID` structure that stores the unique thread identification number. In contrast, for kernel-level threads, we obtain the corresponding pointer from a substructure of the *Kernel Processor Control Region* (KPCR, see Figure 4.3). The KPCR stores processor-related information and is accessible over the `FS` segment register as well (Schreiber, 2001). It is important to keep in mind that all referenced addresses are virtual though. Thus, to process the respective data within Bochs, a virtual-to-physical address translation process has to be completed first for each operation.

When a thread performs a write operation on a memory region that has not been imaged yet, it is checked whether the corresponding thread identification number is included in the watchlist of potentially atomicity-violating candidates. If this is the case, the operation is logged, using a custom binary logger we have implemented for this task. With the help of the generated log file, the upper bound for the degree of inconsistency in the memory image can then be estimated in the next step.

### Integrity

Similarly to the process of measuring the correctness of a memory snapshot, we determine its level of integrity by creating copies of the physical address space at several points in time that are later matched in an analysis phase. In more detail, we duplicate the

state of memory of the guest operating system shortly before the acquisition program is loaded into RAM as well as after the imaging process has finished. The former state is defined to represent the state of memory at time $\tau$, in correspondence to our explanations outlined in Section 3.3.5. To signal the respective point of times, we trigger the `HCALL_LOAD_ACQUISTION_SOFTWARE` and `HCALL_END_LOG` hypercalls that are intercepted by our instrumentation interface. For the first task, a small wrapper utility is running on the guest machine. When the hypercalls are received, the platform temporarily suspends the execution of the guest system and creates a raw memory snapshot. An additional third snapshot is taken before the actual imaging process is started, i.e., *after* the respective solution has been loaded into RAM, but *before* the first page has been acquired. Thus, by comparing the snapshot taken at time $\tau$ with the state of memory at the end of the acquisition process, we can quantify the amount of memory that has either changed over time or remained stable. By the same reasoning, we can roughly estimate the impact of the imaging solution on the system, i.e., contents of RAM that have been overwritten by launching the respective program files, by matching the images taken at time $\tau$ and at the beginning of the acquisition process. Please note, however, that the latter metric is only meaningful when other influencing factors such as the level of concurrent activity have been minimized. Finding better approaches for measuring the impact of an acquisition utility is left for future work.

We have conducted several experiments to quantify the correctness, atomicity, and integrity of forensic memory snapshots created by three different imaging applications. A summary of these activities is subject of the following section.

## 4.3 Evaluation

### 4.3.1 Evaluation Methodology

We have evaluated the performance and quality of three popular memory acquisition utilities, namely *win32dd* (Suiche, 2009b), *mdd* (ManTech CSI, Inc., 2009), and *WinPMEM* (Cohen, 2012b). While all applications have originally been open source, win32dd has been replaced by a closed source variant and is only distributed as part of the *MoonSols Windows Memory Toolkit* to date (Suiche, 2013). We were therefore only able to test the last publicly available version of the source code (`v1.2.1.20090106`).

All experiments were run on a standard off-the-shelf computer system with an Intel i5-650 processor and 8 GB of RAM. The only addition was a solid state drive that was used by the binary logger for performance reasons. For our platform, we reserved a maximum of 2 GB of physical memory. In contrast, for the emulated machine, we chose memory sizes between 512 MB and 2 GB. The latter value represents the maximum amount of memory that is supported by Bochs at the time of this writing, a fact that had unfortunately not been documented in the corresponding user manual and was only discovered after development of the platform had been finished. As the main guest system, we set up a default installation of Microsoft Windows XP (Service Pack 3). To

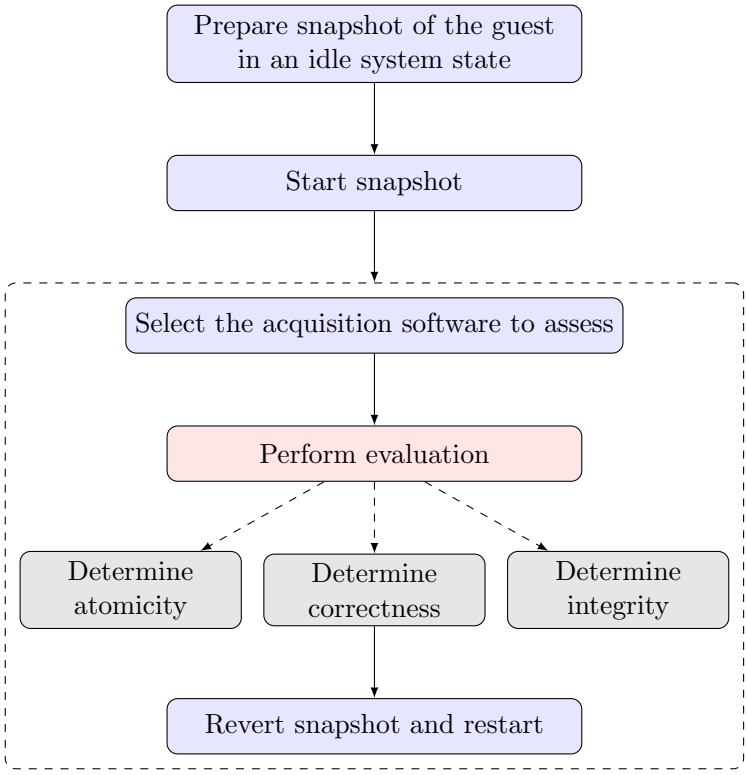



Figure 4.4: Evaluation Procedure for the
Forensic Memory Acquisition Applications

simulate a standard computer as best as possible, the list and configuration of system services was not adapted. However, all tests initially started from an *idle* state, i.e., after the startup process of the operating system and all applications had been fully completed.

For each memory size we considered in our evaluation (`512 MB`, `1,024 MB`, `2,048 MB`), we prepared 30 system *snapshot templates*. Such a snapshot template was created by temporarily suspending the guest system, a feature that is available over the Bochs graphical interface and that permits saving the current state of the processor, memory, and attached devices to hard disk (The Bochs Project, 2013b). Due to programming errors in the emulating engine, however, user input and output are not processed any longer when the simulation is resumed at a later time. For this reason, all image-related activities were automated without requiring any further manual intervention. For each snapshot template, a special hypercall from the host to the guest system indicated the imaging solution to assess. Once a test run had been completed, the snapshot was reverted, and the simulation was restarted from the initial system state for the remaining evaluation candidates. As such, the performance results of the different products with respect to a specific snapshot template were directly comparable. A summary of the evaluation procedure is given in Figure 4.4. In total, we conducted 270 experiments (90

per tested RAM size) for the three acquisition utilities. In dependence of the available memory capacity, between 4.37 GB and 12.37 GB of free hard disk space were required for the created log file.[3] Taking the externally and internally generated snapshots into account as well (5 in total), about 6.87 GB to 22.37 GB of free space were therefore needed for every experiment.

### 4.3.2 Results

#### Correctness

The physical address space of a computer is not only used by the operating system and executed applications, but also shared by hardware devices for *memory-mapped I/O* (MMIO) operations (see Figure 4.5). When generating a forensic snapshot, these areas of memory need to be taken into consideration, too. Specifically, an acquisition solution should identify and zero out MMIO regions in the image file if the respective addresses cannot be accessed. The recommended method for the first task is invoking the undocumented `MmGetPhysicalMemoryRanges` function that indicates the memory structure as it is seen by Windows (Russinovich, 1999). In our tests, both win32dd and mdd initially determined the size of the address space incorrectly. For instance, mdd calculates the size of memory (in pages) based on the output of the `GlobalMemoryStatusEx` function which, unfortunately, does not include information about the MMIO space. Consequently, when iterating through the address space, these regions are ignored, and a smaller snapshot is produced. What is worse, because the image file is written sequentially instead of logically, the offset mapping of subsequently accessible memory areas is corrupted. For analysis techniques that rely on offset interpretation in the evidence extraction phase of the investigation, this may be a significant problem.

With respect to *win32dd*, the error has been fixed in later (closed source) versions of the software (see Suiche, 2009a), the publicly available version of *mdd* is, however, still affected. In order to adequately test both products, we have therefore developed patches that address the issue, so that all considered solutions eventually produced snapshots that truly equaled the size of memory. One peculiarity of Bochs is, however, that the top of the guest's physical address space is always shortened by 16 pages (65,536 bytes). Thus, for a machine with 512 MB of memory for instance, the total number of pages corresponds to 131,056 (536,805,376 bytes) instead to 131,072 (536,870,912 bytes). We believe this may be due to some internal configuration, contacting a Bochs developer did not help shed light on this issue yet though.

The results of our experiments are presented in Table 4.2.[4] As can be seen, all imaging utilities were proven to achieve a correctness rate over 99.3%, i.e., the solutions are

[3] An entry in the log file occupied 12 bytes, i.e., roughly between 391 million (for 512 MB of RAM) and 1.107 billion (for 2,048 MB of RAM) atomicity-related memory operations were observed on average during an imaging process.

[4] All results are average values, based on 30 test runs for each imager and per memory size.

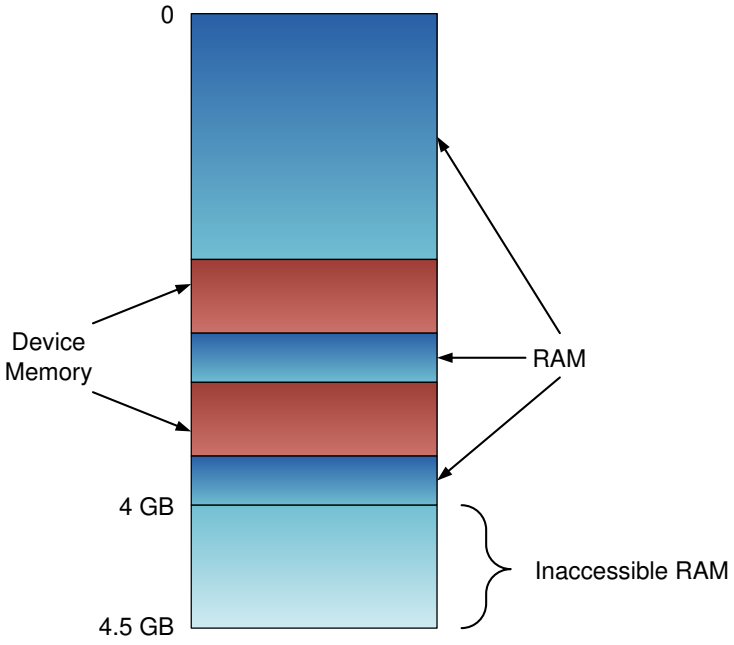



Figure 4.5: Sample Architecture of a Physical Address Space
(Russinovich, 2008)

capable of reliably acquiring the contents of memory. For a small number of pages (see Rows 2, 6, and 10 of Table 4.2), the comparison of the generated snapshot with our external image indicated minor differences. As we have pointed out in Section 4.2.2, these differences lead back to little inaccuracies in our measurement approach, rather than to a malfunction in the imaging software. In most cases, the respective changes were limited to a few bytes per page (typically less than 20). For win32dd, however, more than 25% of one memory page was updated in each test run. We have not investigated the exact reason for this behavior yet, but we assume the location may be used for a program buffer.

If a memory region could not be read successfully, all utilities zeroed out the corresponding parts in the snapshot. Surprisingly, errors were not only signaled for the memory space of hardware devices as expected, but also for various pages of the operating system. Again, the exact reasons for this behavior still need to be examined in more detail. Likewise, it is still unclear at the time of this writing why the execution of WinPMEM led to a significantly higher number of read access violations (98-99 pages) in comparison to its competitors.

| | | **512 MB** | **1,024 MB** | **2,048 MB** |
|---|---|---|---|---|
| Number of pages in the address space | | 131,056 | 262,128 | 524,272 |
| *win32dd* | Number of correctly imaged pages | 130,269 (99.40%) | 260,572 (99.41%) | 521,183 (99.41%) |
| | Number of pages with observed differences | 19 | 20 | 17 |
| | Number of bytes changed in total | 1,446 | 1,484 | 1,388 |
| | Number of indicated read errors (incl. MMIO) | 768 | 1,536 | 3,072 |
| *mdd* | Number of correctly imaged pages | 130,286 (99.41%) | 260,590 (99.41%) | 521,198 (99.41%) |
| | Number of pages with observed differences | 1 | 1 | 1 |
| | Number of bytes changed in total | 1,074 | 1,075 | 1,076 |
| | Number of indicated read errors (incl. MMIO) | 769 | 1,537 | 3,073 |
| *WinPMEM* | Number of correctly imaged pages | 130,188 (99.34%) | 260,492 (99.38%) | 521,100 (99.39%) |
| | Number of pages with observed differences | 1 | 1 | 1 |
| | Number of bytes changed in total | 14 | 14 | 14 |
| | Number of indicated read errors (incl. MMIO) | 867 | 1,635 | 3,171 |

Table 4.2: Results for the Correctness Evaluation of
Different Memory Acquisition Applications

**Atomicity**

In our experiments, the level of potential atomicity violations rapidly increased with the size of installed memory (see Table 4.3). While we monitored potential snapshot inconsistencies between 37.29% and 39.55% of the pages on a machine with 512 MB of RAM, the number of violations jumped up to more than 75% for memory capacities of 2 GB. We believe that the major influencing factor for this steep rise is *time*, assuming the respective load on the system remains constant: As it takes imagers longer to complete their operations on computers with larger amounts of memory, concurrently running applications also have a longer time span to access areas that have already been duplicated and, in the next step, modify regions that are possibly not yet part of the snapshot. In short, with longer imaging periods, it gets obviously more and more difficult to keep the image file free from "smearing" (see also Figure 4.6).

At the time of this writing, it is still an open research problem in what cases and in how far concurrent activity during the acquisition process actually has an impact on the "outcome" of a later analysis. On the other hand, a significant degree of unatomicity is counterintuitive to classic perceptions of "forensic soundness" (see our discussion in

| | | 512 MB | 1,024 MB | 2,048 MB |
|---|---|---|---|---|
| Number of pages in the address space | | 131,056 | 262,128 | 524,272 |
| *win32dd* | Number of pages unaffected by concurrent activity | 81,979 (62.55%) | 115,763 (44.16%) | 133,871 (25.54%) |
| | Number of pages affected by potential atomicity violations | 49,077 (37.45%) | 146,365 (55.84%) | 390,401 (74.46%) |
| *mdd* | Number of pages unaffected by concurrent activity | 82,180 (62.71%) | 113,199 (43.18%) | 127,648 (24.35%) |
| | Number of pages affected by potential atomicity violations | 48,876 (37.29%) | 148,929 (56.82%) | 396,624 (75.65%) |
| *WinPMEM* | Number of pages unaffected by concurrent activity | 79,217 (60.45%) | 113,697 (43.37%) | 127,964 (24.41%) |
| | Number of pages affected by potential atomicity violations | 51,839 (39.55%) | 148,431 (56.63%) | 396,308 (75.59%) |

Table 4.3: Results for the Atomicity Evaluation of Different Memory Acquisition Applications

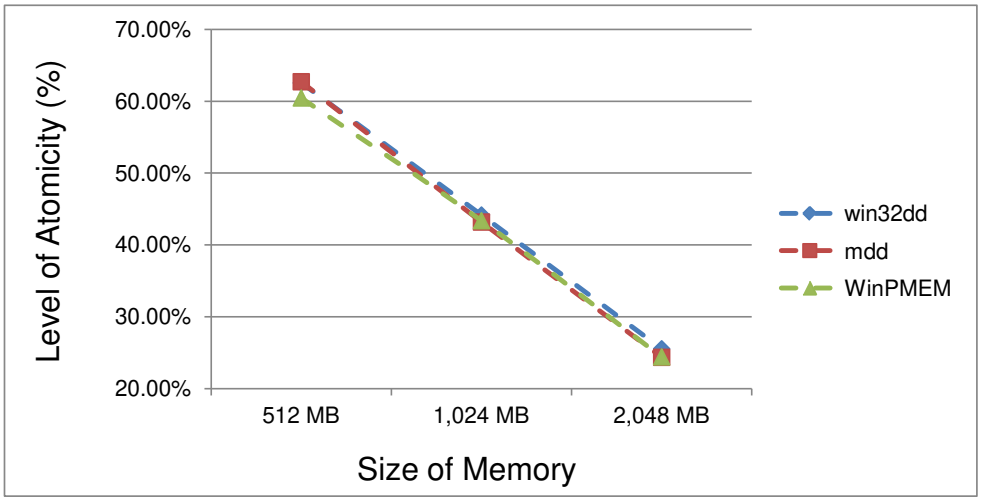



Figure 4.6: Minimum Level of Proven Atomicity for Different Imagers and Memory Sizes

Section 3.4), and it is imaginable that the admissibility of evidence that is clearly based on a highly inconsistent source may be questioned in a trial. Investigators should keep these aspects in mind when using software utilities for the acquisition of a computer's RAM.

**Integrity**

In contrast to the degree of atomicity, the level of integrity the created snapshots satisfied *increased* with growing amounts of available memory. On average, between 27.373% and 28.587% of the data were subject to change in the course of the imaging period on a machine with 512 MB of RAM. Roughly about 47% to 49% of all pages were affected. As opposed to this, on a system with 2 GB of memory, only between 17.905% and 18.447% of the data were modified, even though the number of affected pages slightly went up in most cases. This development was expected, because on a system with constant load but higher memory capacities, proportionally less amounts of space are required for the operating system and running applications. In comparison to former tests by other authors for alternative products and approaches (Walters and Petroni, 2007; Schatz, 2007a), our results are similar. For higher capacities, the level of change is smaller, but with approximately one fifth of the size of the address space still significant. This effect is likely to aggravate again, however, once concurrent activity gets more intense, and the system load reaches its peak.

Regarding the impact of an acquisition utility, between 0.87 and 1.33 MB of memory are changed after loading the respective solution into RAM. As we have pointed out in Section 4.2.2, these values represent an *upper bound* though, rather than an accurate estimation. In sum, however, all tested applications seem quite light-weight and reduce external dependencies on other software components to a minimum. A summary of our experiments and the corresponding results are listed in Table 4.4. The percentage of bytes that remained unchanged in the course of the imaging period is depicted for the different products and memory sizes in Figure 4.7.

| | | **512 MB** | **1,024 MB** | **2,048 MB** |
|---|---|---|---|---|
| Number of pages in the address space | | 131,056 | 262,128 | 524,272 |
| *win32dd* | Number of pages remaining consistent over the imaging process | 68,842 (52.53%) | 138,407 (52.80%) | 265,586 (50.66%) |
| | Number of pages changed during the imaging process | 62,214 (47.47%) | 123,721 (47.20%) | 258,686 (49.34%) |
| | Bytes of memory remaining consistent over the imaging process | 389,382,288 (72.537%) | 869,675,703 (80.999%) | 1,762,918,568 (82.095%) |
| | Bytes of memory changed during the imaging process | 147,423,088 (27.463%) | 204,000,585 (19.001%) | 384,499,544 (17.905%) |
| | Number of pages changed after loading the acquisition program | 1,361 | 1,325 | 1,362 |
| | Bytes of memory changed after loading the acquisition program | 990,792 | 924,691 | 1,024,498 |

*Table continues on the following page.*

| | | | | |
|---|---|---|---|---|
| *mdd* | Number of pages remaining consistent over the imaging process | 69,625 (53.13%) | 140,621 (53.65%) | 266,684 (50.87%) |
| | Number of pages changed during the imaging process | 61,431 (46.87%) | 121,507 (46.35%) | 257,588 (49.13%) |
| | Bytes of memory remaining consistent over the imaging process | 389,867,381 (72.627%) | 872,074,311 (81.223%) | 1,759,805,581 (81.950%) |
| | Bytes of memory changed during the imaging process | 146,937,995 (27.373%) | 201,601,977 (18.777%) | 387,612,531 (18.050%) |
| | Number of pages changed after loading the acquisition program | 1,964 | 1,977 | 1,969 |
| | Bytes of memory changed after loading the acquisition program | 1,287,561 | 1,288,320 | 1,399,336 |
| *WinPMEM* | Number of pages remaining consistent over the imaging process | 66,305 (50.59%) | 137,669 (52.52%) | 266,588 (50.85%) |
| | Number of pages changed during the imaging process | 64,751 (49.41%) | 124,459 (47.48%) | 257,684 (49.15%) |
| | Bytes of memory remaining consistent over the imaging process | 383,347,294 (71.413%) | 861,257,118 (80.216%) | 1,751,276,845 (81.553%) |
| | Bytes of memory changed during the imaging process | 153,458,082 (28.587%) | 212,419,170 (19.784%) | 396,141,267 (18.447%) |
| | Number of pages changed after loading the acquisition program | 1,338 | 1,307 | 1,341 |
| | Bytes of memory changed after loading the acquisition program | 992,730 | 911,888 | 992,381 |

Table 4.4: Results for the Integrity Evaluation of
Different Memory Acquisition Applications

## 4.4 Discussion

### 4.4.1 Black-Box vs. White-Box Testing

As we have already explained, our evaluation platform relies on a *white-box testing* approach, i.e., we have to slightly adapt the source code of the acquisition program and insert several hypercalls that indicate specific events such as the beginning and end of the imaging process. Unfortunately, as we have pointed out in Chapter 2, various solutions are closed source and are frequently commercially distributed. Prominent examples include *Memoryze* (Mandiant, 2011), *FTK Imager* (AccessData, 2012), and *FastDump* (HBGary, 2013). With respect to these applications, we have initially attempted to pursue a *black-box testing* approach, i.e., measuring the quality of an utility despite lacking explicit knowledge of its internal mode of operation. For this purpose, we used

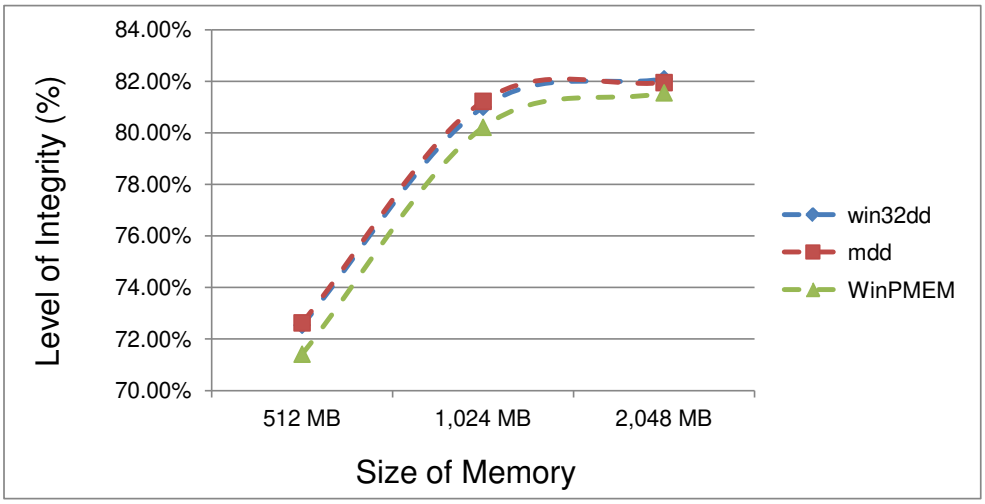



Figure 4.7: Level of Integrity Satisfied for
Different Imagers and Memory Sizes

inline code overwriting techniques to transfer execution to small code caves identified in the respective binary files. The code caves contained instructions to trigger a hypercall and initiate communication with the instrumentation interface. However, the latter method is unreliable for several reasons: First, it is unsure whether sufficiently large areas of free space can be found for the hypercall instructions in every case. Second, in order to get accurate measurement results, it is necessary to insert two hypercalls preferably close to the acquisition routine. This, in turn, would require reverse engineering of a substantial part of the executable though, a process that can be quite cumbersome and is usually legally prohibited.

We also attempted to reduce the number of hypercalls and recognize the imaging process solely based on memory access patterns. In practice, however, this procedure did not prove useful either. We therefore believe that our current implementation of white box testing is best suited for our needs, even though we were capable of evaluating only a small number of products available on the market to date so far. We hope, however, that further vendors will provide us with the source code of their solutions, so that these utilities can be assessed as well.

### 4.4.2 Limitations of the Platform

Our platform still has to deal with a number of weaknesses and limitations at the time of this writing: First, due to some internal program structures, the platform is unfortunately only able to evaluate 32-bit applications. In addition, using Bochs as the underlying emulation engine restricts the size of memory to 2 GB. This is a significant limitation, as even RAM sizes in modern desktop computers are frequently larger, and

it would be particularly interesting to assess imagers with respect to the 4 GB boundary. As we have also pointed out, while we can measure the correctness of a memory image quite accurately, we are only able to quantify the minimum level of atomicity a snapshot satisfies based on the upper bound of potential atomicity violations. Likewise, we can only roughly estimate the impact of the acquisition program by matching the state of memory at time $\tau$ with the state of memory shortly before beginning an imaging operation. For the latter metric, a more suited alternative thus needs to be found.

### 4.4.3 Operational Capabilities of Memory Acquisition Software

With respect to our discussion concerning the characteristics of memory acquisition approaches given in Chapter 2, it is important to balance the operational capabilities of acquisition applications as well as their advantages and weaknesses. Our tests have shown that the performance of the evaluated solutions is quite similar. Forensic investigators may therefore choose the product they find most appealing for their work, assuming the default technique for generating a raw snapshot via the `\\.\Device\PhysicalMemory` section object is not subverted by malicious software, and more sophisticated imaging methods, e.g., via the `MmMapIOSpace` function (see Section 4.1.2), are not required. On the other hand, while imaging software can likely be used in most scenarios (including incident response), a potentially large number of atomicity and integrity violations has to be accepted.

Taking the previous arguments into consideration, relying on approaches that guarantee a higher level of atomicity may thus be a more viable alternative within controlled environments that permit specific, pre-incident preparatory measures. As pointed out in earlier parts of this thesis, suited options are, for instance, the *CrashOnCtrlScroll* functionality of the operating system (Microsoft Corporation, 2011) or the System Management Mode (SMM) approach illustrated by Wang et al. (2011a). To the best of our knowledge though, especially the latter method has not been extensively tested in practice yet.

## 4.5 Further Development and Evaluation Possibilities

As we have explained in Section 4.3.1, all of our tests were initiated from an idle system state. For assessing the quality of an acquisition utility more extensively, it is necessary to simulate different system loads as well and, in particular, monitor the level of atomicity and integrity changes. Experiments should also be conducted on various operating systems, including both server and desktop versions, and on systems that have deliberately been infected with malicious applications.

Using Bochs as the basis for our solution was a suboptimal choice in retrospect. Even though we were able to implement a rudimentary instrumentation interface quite quickly, customizing the emulator to our needs and creating a working prototype of the platform

took significant efforts. This is mainly due to poor user documentation that is incomplete in most parts and only very marginally covers technically more complex concepts. In addition, the readability of the Bochs source code is greatly affected by a vast number of references to program macros. As it is even stated in the official developer manual, many macros have "inscrutable names", and "[o]ne might even go as far as to say that Bochs is macro infested", so that "too much stuff happens behind the programmer's back" (The Bochs Project, 2013c). Last but not least, the functionality of suspending and resuming a simulation is deeply flawed internally. For this reason, many of our original approaches for running a memory acquisition test had to be redesigned. For better long-term maintainability, we therefore recommend porting the platform to a more stable and mature environment such as *QEMU* (Bellard, 2012).

As we have seen, the level of atomicity and integrity a snapshot satisfies can be significantly influenced due to concurrent activity in the course of the imaging period. While violations of these factors may be in direct contradiction to classic perceptions of "forensic soundness" (see Section 4.3.2), the actual consequences of such violations with respect to a subsequent investigation are still mostly unclear. For example, it has yet to be found out in what cases and at what level inconsistency leads to significantly different analysis results. Finding answers to these questions will be an interesting field of research in the near future.

## 4.6 Summary

In this chapter, we have presented a platform for evaluating forensic memory acquisition software with respect to the three factors *correctness*, *atomicity*, and *integrity*. These factors determine the quality of a RAM snapshot and are measured using a *white-box testing* approach, i.e., we must be provided with the source code of the respective solution. With the help of a number of *hypercalls* that are inserted close to image-related code parts, we can then signal important events and operations. A highly customized version of the x86 emulator *Bochs* intercepts these notifications and creates a protocol of the imaging process. In a preliminary study, we have assessed the performance of three acquisition applications, namely *win32dd*, *mdd*, and *WinPMEM*, for memory sizes between 512 MB and 2 GB. For this task, we have analyzed 270 snapshots of systems in an idle state. Our study revealed that not all products were initially capable of generating a copy of the entire physical address space. Even worse, the affected solutions produced image files with mismatching data offsets. Because an analysis of these files would possibly lead to false results in a later investigation, we have patched the different utilities so that the problem was fixed, and correct snapshots were eventually generated.

One interesting observation we made in our evaluation was that the level of atomicity decreased with growing memory sizes. We argued that with larger amounts of memory and, thus, a longer time that is needed to complete the acquisition process, keeping the image file free of inconsistencies gets more and more difficult due to concurrent activity. In contrast, on a system with constant load, proportionally less areas of memory are

subject to change, and the level of integrity a snapshots satisfies increases. In sum, the performance of the tested acquisition utilities (after fixing the respective programming errors) did not significantly differ. This is not surprising, because all solutions internally access the `\\.\Device\PhysicalMemory` section object in kernel space to create a copy of the volatile storage. It is important to keep in mind, however, that these operations may be intercepted by malicious software to either prevent imaging completely or present a modified view of system RAM. Examining the performance of imaging applications within a hostile environment and in the presence of an intelligent adversary that actively pursues anti-forensic measures will thus be an important area for research in the future.

In the following chapter, we will illustrate strategies as well as techniques for analyzing volatile information and extracting valuable pieces of evidence from a snapshot of a computer's RAM.

## Chapter 5

# Forensic Memory Analysis

After a forensic copy of physical memory has been generated with one of the techniques we have illustrated in Chapter 2, an in-depth analysis of the acquired data can begin. Primitive approaches that are described in the literature solely rely on extracting text fragments from an image of a computer's RAM. For this purpose, simple command line utilities such as *strings* or *grep* may be used. In a second step, the captured data may then be examined more closely to retrieve, for instance, stored usernames, passwords, and other valuable artifacts from the snapshot (see Stover and Dickerson, 2005; Zhao and Cao, 2009; Karayianni and Katos, 2011). This methodology is easy to apply, yet it is also noisy, causes a huge overhead, and leads to a large number of false positives. Beebe and Clark (2007, p. S49) argue that "[f]requently, investigators are left to wade through hundreds of thousands of search hits for even reasonably small queries [...] – most of which [...] are irrelevant to investigative objectives".

Even in case basic string searches produce quite accurate results, they typically do not take into account the context of the respective information and, as a consequence, are of limited help to the investigation process. For example, as Savoldi and Gubian (2008, p. 16) point out, "the retrieval of a potentially compromising string (e.g. 'child porn') certainly provides evidence if found in the memory assigned to a process l[a]unched by the user, but it would be likely rejected by jury if that memory belonged to a piece of malware". In addition, Hejazi et al. (2009, p. S123) note that the "existence of unknown sensitive data in the memory is the main important limitation of this method". Thus, a forensic analyst may, e.g., only look for specific keywords, but at the same time, disregard "names, addresses, user IDs, and strings that are not present in the list the investigator is looking for while they are present in the memory dump and are of paramount importance for the investigative case". Taking these aspects into consideration, simple text analysis procedures are therefore not sufficiently suited for forensic investigations. Although more efficient solutions have been developed over the last years, we will not explain them further in the remainder of this thesis, because they rather fall in the research area of information retrieval (IR) and text mining. Good introductions to these topics can be found in the work of Beebe and Clark (2007) as well as Roussev and Richard III (2004) though.

As a viable alternative to string searching tasks, practitioners recommend a more *structured* approach for finding valuable traces in memory. The main focus is thereby set on identifying *what types* of data the snapshot consists of, *how* these types are defined, and *where* they are located. Relevant pieces of information may generally be contained both in the system as well as user address space, either directly in RAM or in the local page

file, and include (Hoglund, 2008; Sutherland et al., 2008):

- the list of currently and previously running processes,
- cryptographic keys,
- artifacts of the system registry,
- established network connections and network-related data such as IP addresses and port information,
- referenced files, and
- system state- and application-related data, e.g., the command history as well as date and timestamp information.

The process for restoring and analyzing the individual artifacts will be the central subject of this chapter.

#### Outline of the Chapter

The remainder of this chapter is outlined as follows: In Section 5.1, we describe fundamental techniques, strategies, and concepts for inspecting memory snapshots. These explanations need to be well understood because they form the foundation for detecting sophisticated malicious applications, a task we will illustrate in more detail in Chapter 6. In Section 5.2, we discuss how memory investigations may be significantly facilitated with the help of powerful analysis frameworks. In particular, we briefly depict the architecture and functionality of the *Volatility Framework* (Volatile Systems, LLC, 2008, 2013a), a popular open source solution that has received broad attention in the forensic community in the past and is regarded as the de-facto standard for memory-based examinations to date. We conclude with a short summary of the most important findings in Section 5.3.

## 5.1 Approaches for Extracting and Analyzing Forensic Artifacts

### 5.1.1 Process Analysis

An essential part of a forensic investigation is verifying the integrity of the target operating system as well as the application environment and distinguishing legitimate software components from possibly infected and compromised program parts. With respect to this, the identification and sound examination of running processes becomes "a basic security task that is a prerequisite for many other types of analysis" (Dolan-Gavitt et al., 2009, p. 2). To obtain the list of currently executing processes, researchers attempt to recover the respective elements either *logically* or *physically*. We will give an overview of both procedures in the following sections.

### Logical Process Enumeration

The primary objective when trying to enumerate processes *logically* is reconstructing a view on resources as they were originally seen by the operating system. For this purpose, an initial `_EPROCESS` object needs to be found. The kernel-level `_EPROCESS` structure serves as an internal representation for a Windows process (Russinovich et al., 2009). A simplified version of the structure is sketched in Figure 5.1. As can be easily seen, an `_EPROCESS` object contains several members that are of immediate importance to an investigator, for instance, two timestamps (`CreateTime`, `ExitTime`) that indicate the creation and destruction of the element. Please note, however, that in dependence of the operating system version and the corresponding service pack level, the layout of the object as well as the offset addressing can significantly vary.

A crucial field of the `_EPROCESS` block is the `ActiveProcessLinks` member, a doubly-linked list that points to the next (respectively, previous) `_EPROCESS` element. Thus, by following the individual pointers, it is possible to reconstruct the process list. Burdach (2005) was first in describing the outlined procedure for memory-based investigations. He also illustrated an algorithm for retrieving the `PsActiveProcessHead` symbol, an important kernel variable that points to the beginning of the `ActiveProcessLinks` list. His suggestions formed the foundation for several early memory analysis programs, e.g., *memparser* (Betz, 2006) or *KnTTools* (Garner, 2007). In a later work, Zhang et al. (2009, 2010) proposed finding the `PsActiveProcessHead` symbol via the *Kernel Processor Control Region* (KPCR). As we have explained in Chapter 4, the KPCR stores processor-specific information and includes a separate block, the *Kernel Processor Region Control Block* (KPRCB) that, for instance, saves CPU-related statistics as well as scheduling information about the current and next thread (see also Russinovich et al., 2009).

A distinct advantage of the KPCR and KPRCB is that they can be found comparatively trivially: In Microsoft Windows XP, they are located at fixed addresses (`0xFFDFF000` and `0xFFDFF120`, respectively). In later versions of the Microsoft Windows product family, however, the base addresses are dynamically computed. Because the KPCR is *self-referencing* (see offset `0x1C` in Figure 5.2), and the KPRCB starts at offset `0x120`, a simple signature can be generated to efficiently locate the structures in a memory image (see Aumaitre, 2009; Schatz, 2010). Once these operations are completed, the process list can then be successfully restored as follows (Barbosa, 2005; Ionescu, 2005): The KPCR contains an interesting field called `KdVersionBlock` that points to the `_DBGKD_GET_VERSION64` structure.[1] This structure, in turn, references the undocumented `_KDDEBUGGER_DATA64` structure. As Dolan-Gavitt (2008b) points out, "[t]he `_KDDEBUGGER_DATA64` structure is used by the kernel debugger to easily find out the state of the operating system" and "contains the memory addresses of a large number of kernel variables", including the `PsActiveProcessHead` variable that, as we have already explained, specifies the top of the `ActiveProcessLinks` list. A summary of these steps is depicted in Figure 5.2.

[1] Cohen (2012a) notes that in recent versions of Microsoft Windows this link is no longer maintained. He therefore proposes an alternative algorithm for directly finding the parent structure of the `PsActiveProcessHead` symbol in memory, based on a simple signature for a specific member.

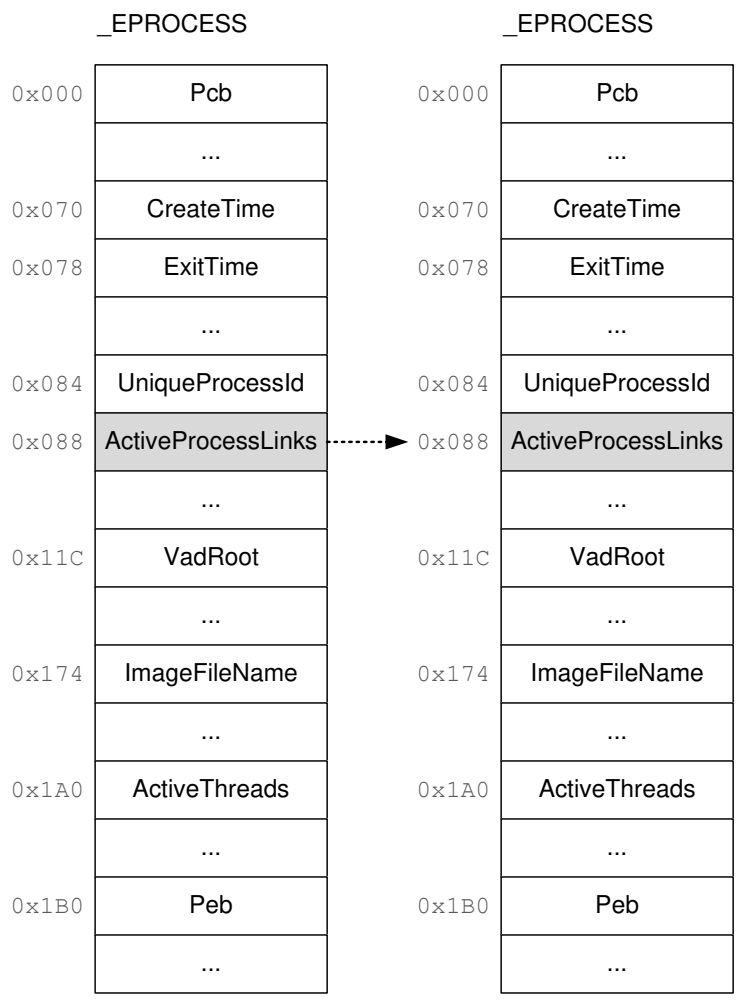



Figure 5.1: Simplified Structure of the `_EPROCESS` Block

### Physical Process Enumeration

One of the major drawbacks of the previously described methods is that member variables such as the `ActiveProcessLinks` field of an `_EPROCESS` block can be subject to manipulation. Especially more sophisticated malicious applications frequently hide their traces more carefully and avoid detection by actively removing their components from the respective system lists. These types of attack are known as *Direct Kernel Object Manipulation* (DKOM) (Aquilina et al., 2008; Ligh et al., 2010), and we will see several examples for kernel-level rootkits that implement such compromising techniques in Chapter 6.

The effects of a DKOM attempt are illustrated in Figure 5.3: In the upper picture, three processes are shown that are correctly linked with each other. The second process in

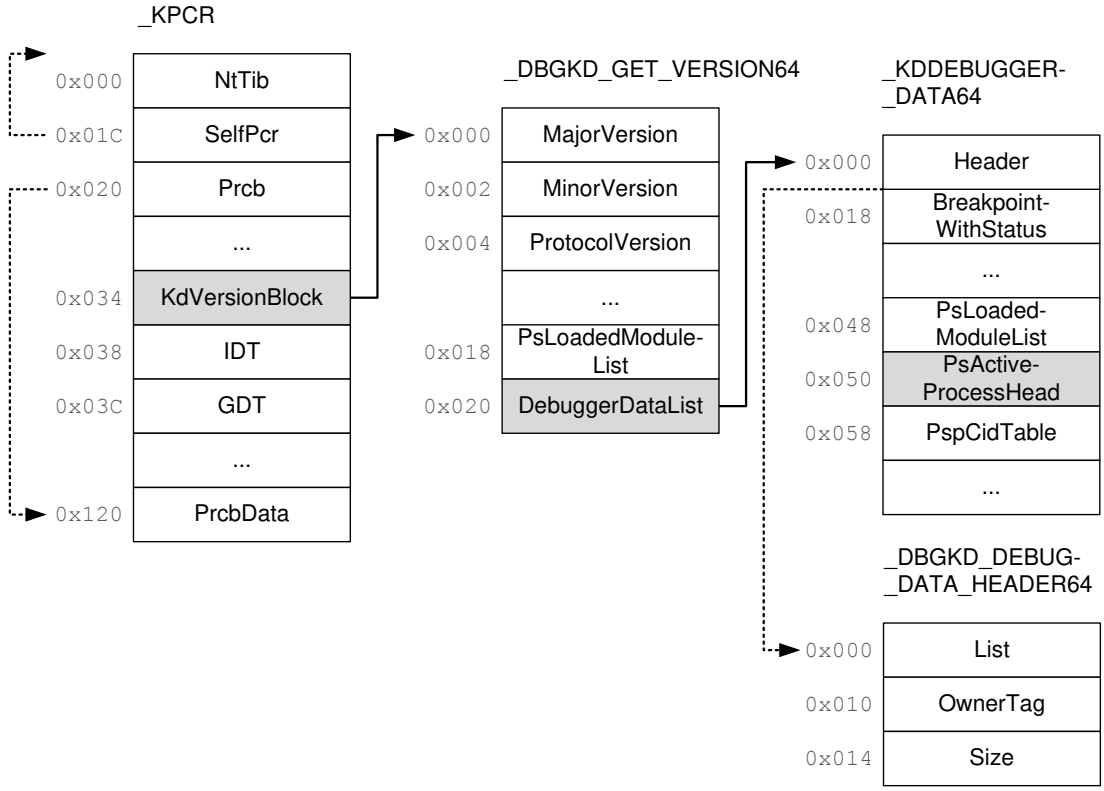
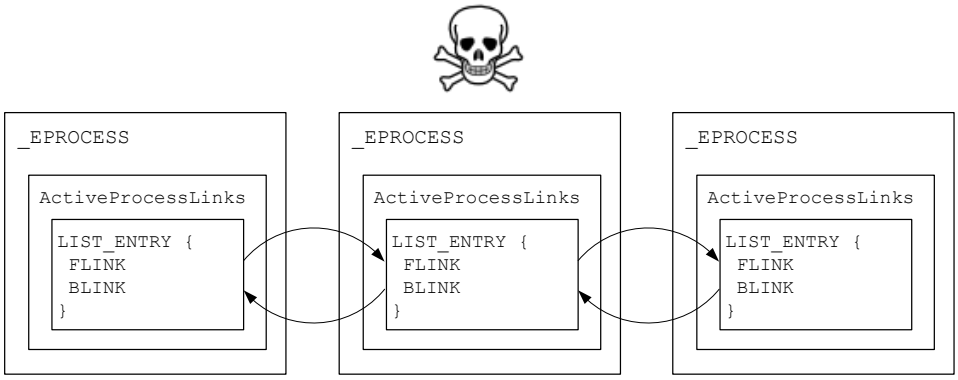


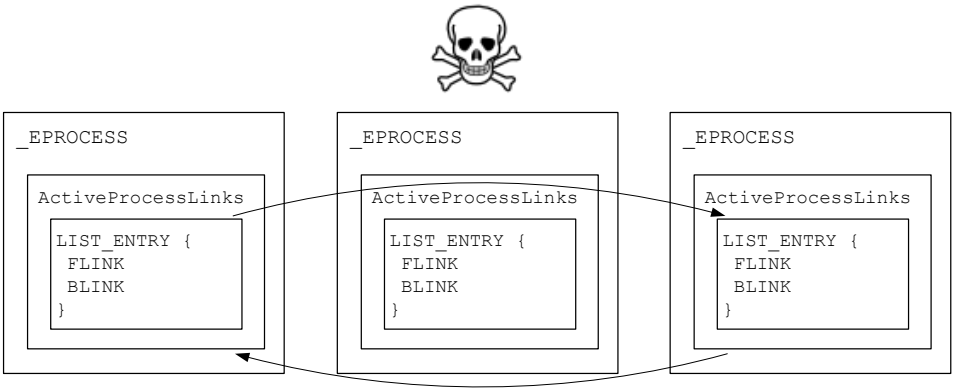

Figure 5.2: Retrieving the `PsActiveProcessHead` Symbol via the Kernel Processor Control Region (KPCR)

the middle is defined to represent a malicious program, as indicated by the skull icon. By modifying the forward (`FLINK`) and back (`BLINK`) pointers of the adjacent processes as shown in Figure 5.3b, however, the malicious program is unlinked from the process list and, thus, effectively hidden from standard system maintenance applications such as the *Task Manager*. Because scheduling is performed on a per-thread basis (Russinovich et al., 2009), the malicious code does still get executed though. In order to cope with these issues, Schuster (2006d) has proposed *physically* scanning the acquired image for occurrences of `_EPROCESS` blocks. For this task, a signature-based scanner is used that identifies the unique appearance of a process. In Listing 5.1, a sample signature is depicted. In the given example, it is checked that the page directory is aligned at a page boundary, the control structures of the attached thread are contained in kernel space, and the fields of the `_DISPATCHER_HEADER` structure match pre-defined values.[2] When the physical memory snapshot has been scanned, the generated result set can be compared with the output of the standard process list. Any differences and anomalies that are

[2] The `_DISPATCHER_HEADER` structure is part of any *synchronizable* object, i.e., an object such as a process or thread that "can be waited for" (Schuster, 2006b).




(a) Structure of the Process List Before a DKOM Attack




(b) Structure of the Process List After a DKOM Attack

Figure 5.3: Manipulation of a Process List with Direct Kernel Object Manipulation (DKOM)

discovered in this step potentially indicate the presence of a malicious program and therefore need to be inspected in more detail. Similar strategies are also suggested by Walters and Petroni (2007) as well as Carr (2006).

It is important to emphasize that in contrast to the logical enumeration method we have described in the previous section, the signature-based technique can not only recover resources that are currently in use, but also objects whose life cycle has already ended but are still present in memory. Likewise, the approach can not only be employed for finding processes but other elements such as network connections, system services, or kernel drivers as well. We will illustrate the procedure for restoring these types of artifacts in a later part of this chapter. On the other hand, it is crucial to point out that the efficiency of the scanning process significantly depends on the quality of the applied

```
1  // Ensure the Page Directory is aligned at a page boundary
2  PageDirectoryTable !=0
3  PageDirectoryTable % 4096 == 0
4
5  // Ensure the control structures of a process's thread are
6  // located in kernel space
7  (ThreadListHead.FLink > 0x7FFFFFFF) &&
8  (ThreadListHead.Blink > 0x7FFFFFFF)
9
10 // Ensure the object corresponds to a process (type 3) and
11 // has a fixed size
12 _DISPATCHER_HEADER.Type == 0x03
13 _DISPATCHER_HEADER.Size == 0x1b
```

Listing 5.1: Example of a Process Signature (Schuster, 2006d)

signatures. As Walters and Petroni (2007, p. 14) argue, "a reliable pattern may rely on characteristics of an object that are not essential". Consequently, an adversary may change the value of a *non-essential* structure member without affecting the stability of the underlying operating system. For instance, regarding the signature shown in Listing 5.1, the `Size` variable is a non-essential field and can be set to zero. Thereby the scanner is circumvented, and the respective application is concealed.

To prevent such scenarios, Dolan-Gavitt et al. (2009) have created so-called *robust* signatures that are solely based on structure members that are critical to system functionality. A manipulation of such a variable leads to an automatic system crash and, thus, renders an attack useless. With respect to the `_EPROCESS` structure, 72 fields (out of 221) suffice the latter requirement and, therefore, form strong candidates for a process signature.[3] However, Haruyama and Suzuki (2012) have recently proven that many memory analysis tools available on the market to date have not adequately considered these valuable insights yet.

### 5.1.2 Cryptographic Key Recovery

As we have argued in the introduction of the thesis, due to high availability of (free) encryption technologies, security professionals are likely to encounter an increasing number of hard disks and other storage media in the future that are especially secured against unauthorized access. If a suspect is unable or unwilling to share the respective passphrase with an investigator in such a case, restoring the respective information from volatile memory may therefore become a central task in the course of a forensic analysis.

Different methods are proposed in the literature for cryptographic key recovery: Hargreaves and Chivers (2008) describe a linear memory scanning technique that cycles

[3] After performing a fuzzing test, the number of strong signature candidates was reduced from 72 fields to 43 (Dolan-Gavitt et al., 2009).

through RAM one byte at a time, using a block of bytes as the possible decryption key for the volume in question. The brute force-oriented procedure does not require a deep understanding of the underlying operating system and, thus, can be easily generalized according to the authors. However, it cannot be directly applied if the key is split, i.e., is not stored in a contiguous pattern in memory. Shamir and van Someren (1999) seek for sections of high entropy to locate an RSA key within "gigabytes of data". The solution exploits the mathematical properties of the cryptographic material. In contrast, the attack described by Klein (2006a) is based on the observation that both private keys and certificates are stored in standard formats. By constructing a simple search pattern, the secret information can be easily extracted from a snapshot of a computer's RAM. A pattern-like approach is also pursued by Kaplan (2007). His idea stems from the fact that, for reasons of security, cryptographic keys should not be stored on hard disk and, thus, are likely to be found in the *non-paged pool* of the operating system. The non-paged pool is a region of virtual memory in the system address space that is never paged out to secondary storage (Russinovich et al., 2009). One characteristic of pool memory is that it may be associated with an identifier, the so-called *pool tag* (Microsoft Corporation, 2013c). As Kaplan (2007, p. 18) points out, "a cryptosystem-specific signature, consisting of the driver specific pool tag and pool allocation size are all that is necessary to extract pool allocations containing key material from a memory dump with an acceptably small number of false positives".

Walters and Petroni (2007) outline a different concept that relies on an analysis of publicly available source code. They identify internal data structures that are responsible for holding the master key. As Maartmann-Moe et al. (2009, p. S133) point out, Walters and Petroni "do, however, not describe how to locate the different structures in memory, and neither do they discuss the fact that some of these may be paged out, thereby breaking the chain of data structures that leads to the master key if only the memory dump is available for analysis". As an alternative, Halderman et al. (2008) therefore suggest parsing a computer's memory for key schedules. As we have already explained in Section 2.2.7, the authors leverage remanence effects in DRAM modules and launch a *cold boot* attack on the target machine. After loading a custom operating system, the volatile data is extracted, the key material is retrieved, and the hard drive is automatically decrypted.

The performance of this process has been improved in the works of Heninger and Shacham (2009). Likewise, Tsow (2009) presents an algorithm that is capable of recovering cryptographic information from memory images that are significantly decayed and is magnitudes faster than the original method. Maartmann-Moe et al. (2009) extend this research on additional ciphers and illustrate the vulnerability of several well-known whole-disk and virtual-disk encryption utilities. Even though cold boot attacks have proven successful for recovering encrypted information, it is important to note that these methods can be effectively mitigated by implementing cryptographic algorithms entirely on the microprocessor. A corresponding proof-of-concept application has been published by Müller et al. (2010) and has been further developed in various other projects (see Müller et al., 2011, 2012).

### 5.1.3 System Registry Analysis

The Windows *registry* is a central, hierarchically-organized repository for configuration options of the operating system and third party applications (Microsoft Corporation, 2012b). It is internally structured into a set of so-called *hives*, i.e., discrete, treelike databases that hold groups of registry keys and corresponding values (see also Russinovich et al., 2009). Most registry hives, for instance, `HKLM\SYSTEM` or `HKLM\SOFTWARE`, are persistently stored in the `system32\config` folder of the operating system.[4] However, a number of *volatile* hives, e.g., `HKLM\HARDWARE`, are solely maintained in RAM and are created every time the system is booted. While the examination of on-disk registry data is a quite established procedure for finding possible pieces of evidence in the course of a forensic investigation (see Carvey, 2011), the only memory-based approach we are aware of has been documented by Dolan-Gavitt (2008c). In the following, we give a short overview of the prevailing techniques used in this work.

#### Architecture of a Registry Hive

A registry hive consists of a so-called *base block* and a number of *hive bins* (hbins). The base block with a fixed size of 4 KB defines the start of the hive and contains a unique signature (`regf`) as well as additional meta information. The latter include a time stamp that saves the time of the last access operation, an index to the first key, i.e., the *RootCell*, the internal file name of the hive, and a checksum (see Figure 5.4). A hive bin is typically 4 KB wide (or a multiple of it) and serves as a container for *cells* that, in turn, store the actual registry data. Thus, to read a certain registry key or value from RAM, it is necessary to locate the correct hive and cell first. With respect to the former task, it is possible to create a hive-specific signature and scan the memory snapshot, in correspondence to our explanations regarding the physical restoration of processes described in Section 5.1.1: Internally, a hive is represented by a `_CMHIVE` structure which is allocated from the *paged* pool of the operating system. The `_CMHIVE` structure is referenced by a specific tag (`CM10`) and embeds a substructure `_HHIVE`. The latter contains a member variable at offset `0x000` that stores the constant string value `0xbee0bee0`. With the help of this information, a unique search pattern can be generated, and the hive can be easily found in memory. Likewise, by following the respective pointers saved in the `HiveList` field (offset `0x224`), the remaining hives loaded into RAM may be enumerated. Please note that these steps are similar to the *logical* process reconstruction method via the `ActiveProcessLinks` list outlined in Section 5.1.1.

#### Locating Registry Keys and Values in Memory

Retrieving pre-defined registry keys or values from a memory image is slightly more complex: Dolan-Gavitt (2008a) notes that because "space for the hives is allocated out

[4] User-specific settings are saved in the file `NTuser.dat` that is located in the `%SystemDrive%\Documents and Settings\<username>` folder.

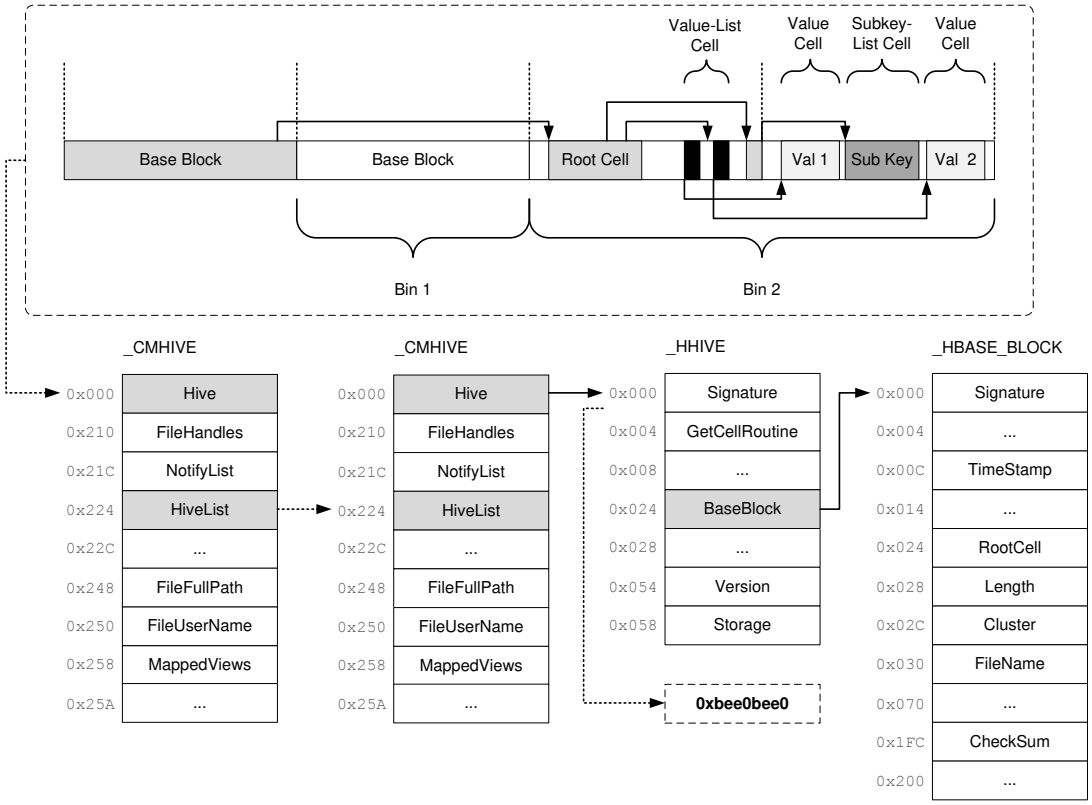



Figure 5.4: Structure of a Registry Hive
(Based on Russinovich et al., 2009, p. 271)

of the paged pool, there is no way of guaranteeing that the memory allocated for the hive will continue to be contiguous". For this reason, a slightly different strategy needs to be pursued: A registry cell is internally referenced by a *cell index*. As indicated in Figure 5.5, a cell index is split into four components. Similarly to the virtual-to-physical address translation process explained in Chapter 2, each component must be analyzed to obtain a virtual address for the cell in question. The first element of a cell index, i.e., a one-bit flag, determines whether the cell is part of a *stable* hive on disk or refers to a *volatile* container in memory. Depending on the value of this flag, a different hive *storage map* is used. The storage map defines a starting point for the translation process we will describe in more detail in the following and is saved in a member of the previously outlined `_HHIVE` substructure (offset `0x058`, see also Russinovich et al., 2009). Thus, once the storage map of a hive has been found, we can examine the first 10 bits of the cell index to search the corresponding entry in the *hive map directory*. This directory saves pointers to 1,024 different *cell map tables*. A cell map table, in turn, consists of 512 pointers to registry blocks that contain the individual registry cells with the respective data. Consequently, to read a specific key or value, it is first necessary to follow the pointer from the hive map directory to the cell map table and subsequently locate the

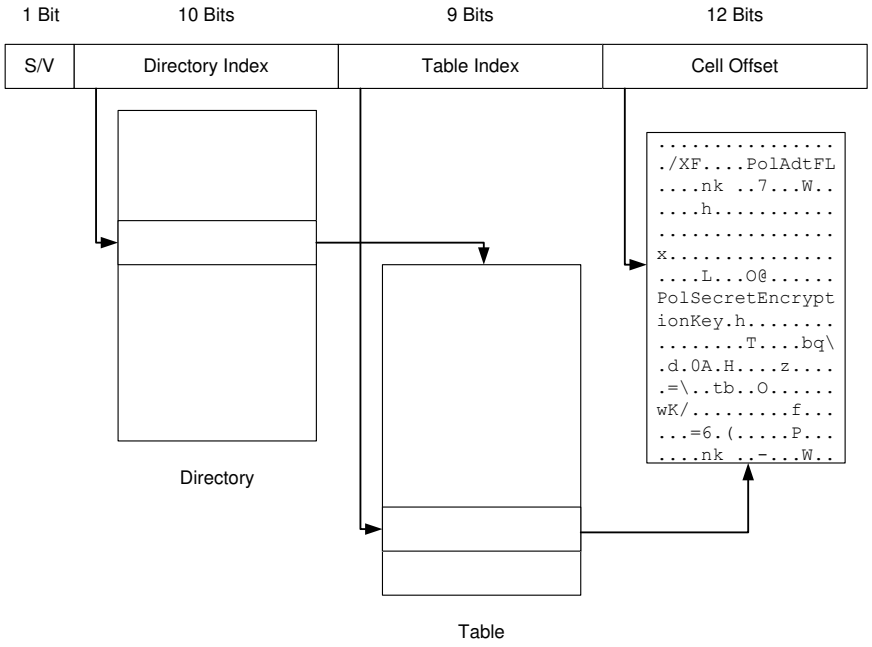


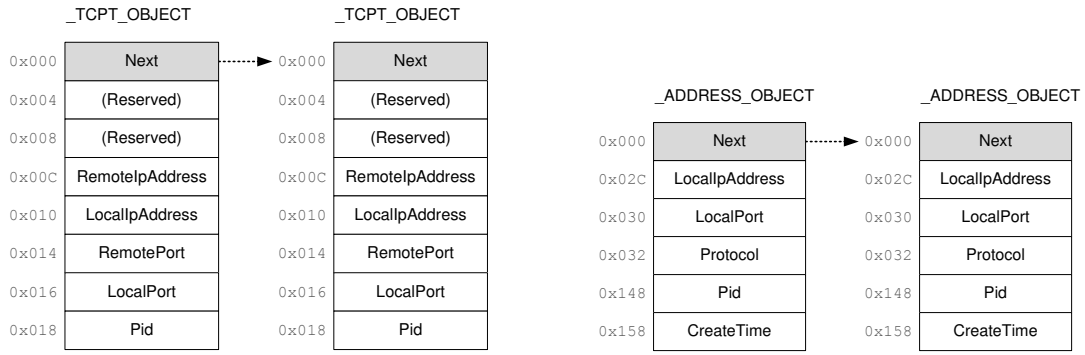

Figure 5.5: Structure of a Cell Index
(Dolan-Gavitt, 2008c, p. S28)

correct entry in the table to identify the target bin. For the latter purpose, the next 9 bits of the cell index must be parsed. Using the remaining 12 bits of the cell index in the last step, the cell with the desired registry data can finally be discovered.

The concepts outlined above have been integrated into the *Volatility Framework*, a powerful memory analysis application we will describe in more detail in Section 5.2. It is important to emphasize, however, that portions of a hive are only transferred into RAM when they are needed (Russinovich et al., 2009). Therefore, as Dolan-Gavitt (2008c, p. S30) points out, it is possible that "parts of the registry may have never been brought into memory in the first place" and, thus, "it cannot be assumed that the data found in memory is complete". Forensic analysts should keep these aspects in mind when attempting to investigate registry-related artifacts.

### 5.1.4 Network Analysis

Malicious applications frequently open pre-defined ports on a machine to permit an adversary executing arbitrary commands, disabling security mechanisms, or uploading further attack tools (Aquilina et al., 2008; Ligh et al., 2010). Inspecting network connections and monitoring inbound or outbound network traffic is therefore frequently an integral part of a forensic analysis. A large number of applications have been developed in the past for the latter tasks, for example, *TCPView* (Russinovich, 2011) and




(a) `_TCPT_OBJECT` List (b) `_ADDRESS_OBJECT` List

Figure 5.6: Structure of the `_TCPT_OBJECT` and `_ADDRESS_OBJECT` List

*FPort* (Foundstone, Inc., 2000) that are more suited for incident response situations, or *PyFlag* (Cohen, 2008) that especially aims at post-mortem investigations. Since network information is typically maintained in RAM as well, memory-based examinations are also well capable of extracting valuable pieces of evidence and helping practitioners gain a better picture of an incident. Network resources can thereby be reconstructed either *logically* or by *physically* scanning the memory image, similarly to the process recovery-related methods described in Section 5.1.1. In the following, we give a brief overview of these concepts.

### Logical Reconstruction of Network Resources

To logically restore established TCP and UDP connections from a suspicious system, the two internal lists `_TCPT_OBJECT` and `_ADDRESS_OBJECT` need to be parsed. The lists are defined in the `tcpip.sys` driver file but are officially undocumented. However, their format has been successfully reverse engineered by several authors in the past (Ligh et al., 2010; Okolica and Peterson, 2010b). The structure of the two lists is depicted in Figure 5.6a and 5.6b.[5] As can be seen, objects are linked via the `Next` member, i.e., a pointer to the subsequent element. Thus, by following the individual references until an element is reached that points to *zero* (see Figure 5.7), the list of open sockets and network connections can be retrieved. The top of the `_TCPT_OBJECT` and `_ADDRESS_OBJECT` list is thereby indicated by two global symbols that are located at specific offsets in the `tcpip.sys` driver file[6].

Please note that the previous descriptions only apply to Microsoft Windows XP operating systems. On Microsoft Windows 2003, the offsets of member variables are different,

[5] For more information, please see the `tcpip_vtypes.py` file in the `volatility\plugins\overlays\windows` folder of the *Volatility Framework* (Volatile Systems, LLC, 2013a).

[6] For more information on the individual offsets, please refer to the `network.py` file in the `volatility\win32` folder of the *Volatility Framework* (Volatile Systems, LLC, 2013a).

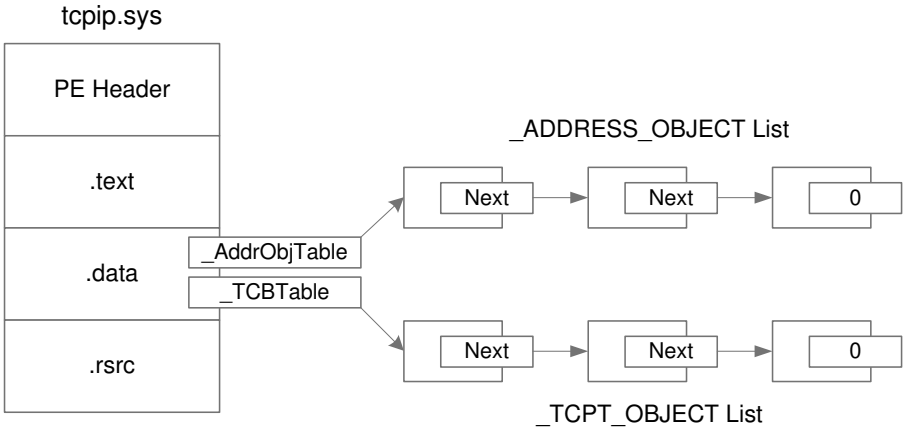



Figure 5.7: Enumerating the List of Socket and Connection Objects (Ligh et al., 2010, p. 677)

and in later Microsoft Windows versions, the respective structures have even been replaced by other constructs. More information for restoring network resources on these platforms can be found in the work of Okolica and Peterson (2011) as well as Wang et al. (2011b). It is also worth mentioning that the `_TCPT_OBJECT` and `_ADDRESS_OBJECT` lists are not as severely affected by Direct Kernel Object Manipulation attacks as, for instance, the process-related elements we have illustrated in Section 5.1.1. As Ligh et al. (2010) point out, it is possible to modify both lists on the one hand, yet these operation lead to a disruption of the communication process, and connections can no longer be initiated. From a forensic point of view, logical network enumeration methods may therefore be seen as sufficiently reliable at the time of this writing and permit reconstructing a genuine view of network activities.

### Physical Reconstruction of Network Resources

In addition to the techniques outlined in the previous section, network resources can also be *physically* recovered. A major benefit of this approach is that it does not only offer finding objects that are currently in use, but also elements that are not active anymore but have not yet been overwritten. A signature-based solution for this task is suggested by Schuster (2006c). Schuster proposes scanning the non-paged pool of the operating system to find allocations for listening sockets. His idea relies on the fact that, when reserving memory in kernel space, the respective regions are marked with a special identifier, the *pool tag*, to facilitate later debugging and code verification (see the description of the `ExAllocatePoolWithTag` function (Microsoft Corporation, 2013c) and our explanations regarding kernel pools given in Section 5.1.2). After reverse engineering the relevant parts in the `tcpip.sys` driver file, Schuster argues that 368 bytes of memory are required to set up a network socket. The corresponding allocations are identified by

the unique string `TCPA` (in little-endian format).[7] With the help of these findings, the signature body can be easily created, and the RAM image can be searched for socket definitions. By following a similar procedure, the list of open network connections can be retrieved, too.

### 5.1.5 File Analysis

In the course of a forensic analysis, a crucial task for investigators may not only be examining the individual running processes on the target machine (see Section 5.1.1), but also verifying other software artifacts that are accessed and referenced by the respective programs. Of particular importance is, for instance, inspecting the list of *dynamically loaded libraries* (DLLs), because adversaries frequently inject malicious modules in the address space of legitimate applications in order to escalate their privileges or further compromise the system environment. We will see examples for such attacks in Chapter 6 of this thesis. To reveal these types of manipulations, security professionals may attempt to reconstruct shared libraries either *logically* (i.e., restoring the original view of the operating system) or *physically* (i.e., scanning the memory image with the help of signatures), in correspondence to the process- and network-related measures we have described in the previous sections of this chapter.

#### Logical File Recovery

The set of dynamically loaded libraries that are associated with a process can be derived from its *Process Environment Block* (PEB), i.e., a user-space structure that "contains information needed by the image loader, the heap manager, and other Windows system DLLs" (Russinovich et al., 2009, p. 341). As can be seen in Figure 5.8, the PEB contains a member `Ldr` of type `_PEB_LDR_DATA`. This data structure contains three doubly-linked lists, `InLoadOrderModuleList`, `InMemoryOrderModuleList`, and `InInitializationOrderModuleList`, that store the same information, yet in different order. Each element in one of these lists is of type `_LDR_DATA_TABLE_ENTRY` and saves a description for a shared library with its full name, size, and base address. Thus, by enumerating and comparing the different lists, a maliciously injected DLL can possibly be identified.

With respect to the previous explanations, it is critical to keep in mind that the PEB and the associated substructures are located in user mode and, thus, can be easily modified as various rootkit authors have demonstrated (Darawk, 2005; Kdm, 2004). For this reason, it is generally recommended to match results with the list of memory-mapped files as they are defined in the *Virtual Address Descriptor* (VAD) tree. Virtual Address Descriptors are kernel-level data structures "the memory manager maintains (...) to keep track of which virtual addresses have been reserved in the process's address space" (Russinovich

[7] According to Wang et al. (2011b), the tag for socket-related memory allocations has changed in Microsoft Windows 7 operating systems to `TCPE`.

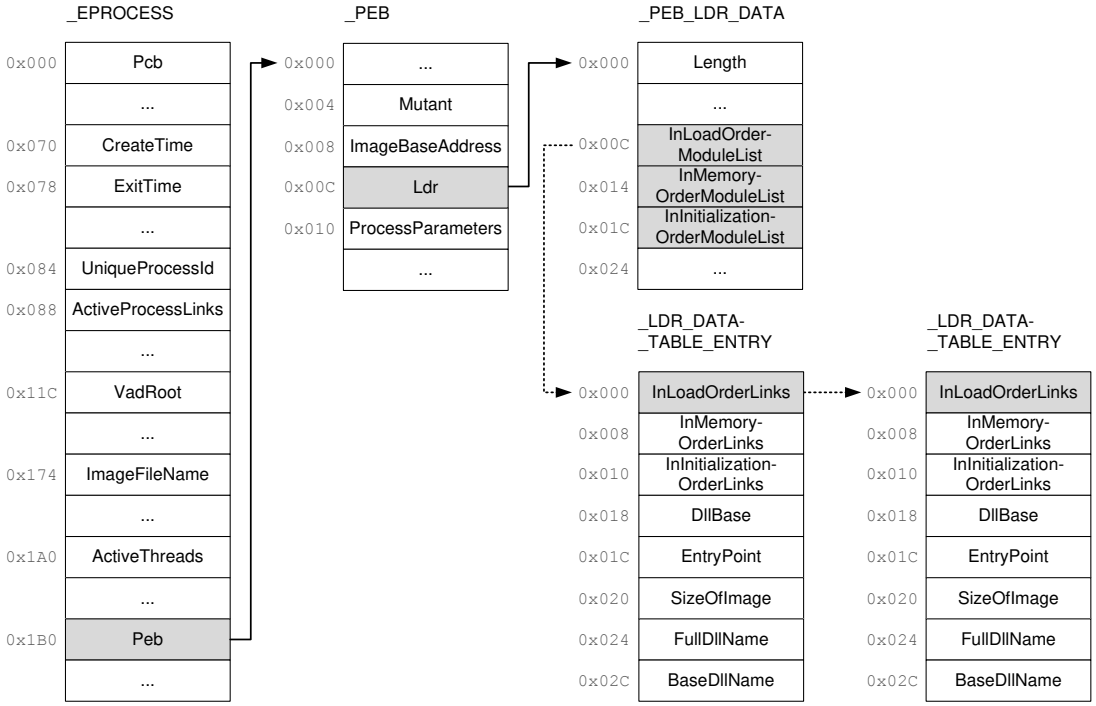



Figure 5.8: Reconstruction of the List of Loaded Libraries via the Process Environment Block (PEB)

et al., 2009, p. 788). Each time address space is allocated, a VAD with the respective start and end addresses as well as additional access and control flags is created and added to a self-balancing tree. A node in this tree is associated with a unique pool tag and is either of type `_MMVAD_SHORT` ("VadS"), `_MMVAD` ("Vad"), or `_MMVAD_LONG` ("Vadl") (Dolan-Gavitt, 2007a; van Baar et al., 2008). The latter two store a pointer to a so-called *control area* that, in turn, points to a `_FILE_OBJECT` structure (see Figure 5.9). In combination with several status-related flags, this structure holds the full name and size of a memory-mapped file. Thus, by traversing the VAD tree from top to bottom and following the individual `_CONTROL_AREA` and `_FILE_OBJECT` references, the list of loaded modules can be restored. In spite of these features, it is important to emphasize that the VAD tree may also be subject to manipulation, assuming an adversary possesses system privileges. In this case, it is, for instance, possible to remove a node from the tree and thereby hide the referenced memory regions without affecting their accessibility (see Dolan-Gavitt, 2007a). Investigators are therefore advised to carefully check their analysis results for inconsistencies. In Chapter 6 of this thesis, we will present a software application that facilitates some of these steps.

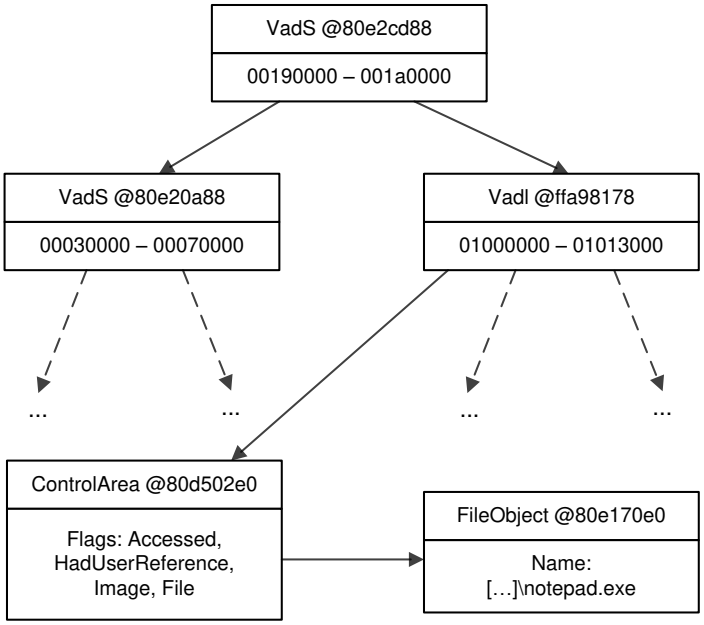



Figure 5.9: Virtual Address Descriptor (VAD) Tree
(Dolan-Gavitt, 2007a, p. S63)

### Physical File Recovery

As an alternative to the previous methods, occurrences of `_FILE_OBJECT` structures can also be found directly in a memory image. For this purpose, the snapshot needs to be scanned with a unique signature. Because file objects are created in the *non-paged* pool of the operating system, are identified by the `File` pool tag, and have a minimum block size of `0x98` bytes (Schuster, 2009b), this is a comparatively easy task. However, one drawback when solely searching for instances of `_FILE_OBJECT` structures is that investigators do not obtain any information concerning the parent process. For this reason, a file object must be linked with the corresponding `_EPROCESS` structure in a second step: As can be seen in Figure 5.10, the `_FILE_OBJECT` structure is immediately preceded by an `_OBJECT_HEADER` structure. This structure contains a member `HandleInfoOffset` that points to an offset before the object header. At this memory location, the desired `_EPROCESS` structure can be found or, alternatively, a reference to an *object handle count database* that contains pointers to several `_EPROCESS` structures in case a handle to the file is kept open by multiple processes (Schuster, 2009a; Noone, 2009).

## 5.1.6 System State- and Application-Specific Analysis

A memory image frequently also includes system state-related information that may be of great benefit to an investigator. Especially the `_EPROCESS` block we have illustrated

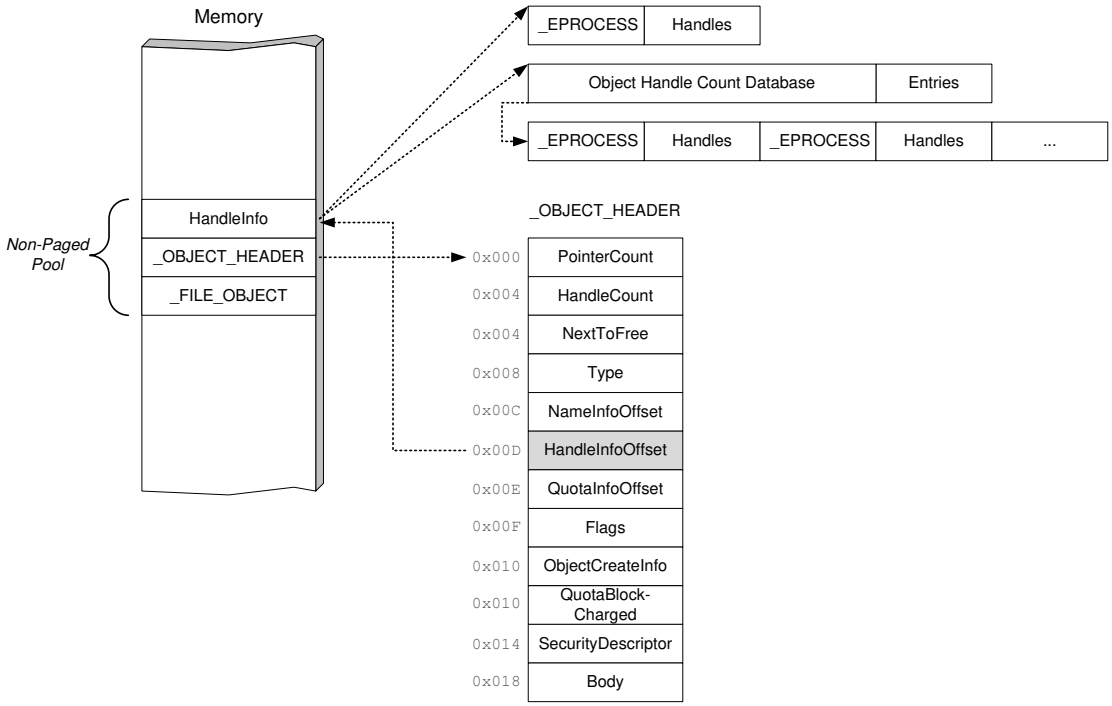



Figure 5.10: Linking File Objects to Processes

in detail in Section 5.1.1 is a source of valuable data. For instance, the `StartTime` and `ExitTime` members indicate the start and respective end time of a process and can be parsed to create forensic timelines. In addition, the periods an application has spent in system and user mode can be derived from members of the `_KPROCESS` block, a substructure of the `_EPROCESS` environment. Schuster (2008b) has proven that these types of artifacts may be recovered from RAM even after a program has terminated for more than 24 hours. Furthermore, with the help of the `Token` member, it is possible to reconstruct the *security context* of an application. As Russinovich et al. (2009, p. 473) explain, "a security context consists of information that describes the privileges, accounts, and groups associated with the process or thread". Of particular interest is the list of user and group *security identifiers* (SIDs) that eventually reveal the name of the user and corresponding group account the executable was run as. More details about this procedure can be found in the work of Dolan-Gavitt (2008d) and Gurkok (2012).

A different research focus is set by Stevens and Casey (2010). They dissect the structure of the `DOSKEY` utility that is integrated into the command shell and permits editing past commands as well as displaying the command history. The latter is solely maintained in memory and is only accessible as long as the command prompt is open. As Stevens and Casey (2010, p. S58) point out, "[i]n practice, this makes recovering the command history difficult due to its volatile nature and the low likelihood of finding an open command window during an investigation", even though major parts "may be recoverable from

memory for some time after the window has been closed". The authors generate a unique signature for various `DOSKEY` elements and succeed in restoring command history objects from a number of reference data sets, including intact lists of entered commands. As these "might contain the only retrievable traces of a deleted file or suspect activity", such types of examinations support other system state-oriented investigation methods and "can provide significant context into how and what occurred on [a] system" (Stevens and Casey, 2010, p. S57).

### Application Analysis

While most approaches we have described so far target operating system structures, "one of the issues currently faced in the analysis of physical memory is the recovery and use of application-level data" (Simon and Slay, 2009, p. 996). As Simon and Slay (2010) argue, these steps become necessary due to the increasing spread of anti-forensic technologies and a trend towards online and network applications. Research in this area still remains sparse at the time of this writing though. This fact may possibly be attributed to the significantly lower lifespan of user-space data (Solomon et al., 2007) and, thus, smaller opportunities for retrieving valuable pieces of evidence in time. Applications that have been examined in greater depth mainly comprise instant messaging and Voice over IP (VoIP) telephony software (e.g., see Gao and Cao, 2010; Simon and Slay, 2010, 2011). In contrast, Olajide and Savage (2011) have investigated memory traces of standard office programs, while Hauenstein and Vömel (2013) have concentrated on finding remnants of social networking sites in RAM.

In spite of these insights, White et al. (2012, p. S3) criticize that there is a general "lack of methodologies for understanding and interpreting (...) application data", making it impossible "to extract useful information from the memory of an application (...) without first undertaking significant reverse engineering of the application itself". The authors therefore propose rather conducting "detailed analyses of metadata sources that can be used to describe the purpose of user space allocations". More precisely, they perform an in-depth examination of a process's Virtual Address Descriptor (VAD) tree as well as the Process and Thread Environment Block (PEB, TEB). As we have explained in Section 5.1.5, the kernel-level VAD tree describes the range of memory regions that are occupied by a process. The PEB and TEB, on the other hand, store context- and state-specific information in user space. By correlating the respective structures, White et al. are not only able to enumerate the list of memory-mapped files, but also recover substantial parts of the process's execution environment, e.g., its stack and heap, its runtime parameters, and data about user interface-related elements. Thereby, "the search space when looking for specific information" can be tremendously reduced (White et al., 2012, p. S10).

## 5.2 Framework-Based Memory Analysis

With respect to the approaches and techniques we have outlined in the previous section, various memory analysis utilities have been developed in the past that sophistically solve a certain problem (e.g., see Betz, 2006; Dolan-Gavitt, 2007b; Schuster, 2008c). However, many of these utilities were originally not designed with a holistic forensic process in mind, but rather as a proof-of-concept demonstration. As a consequence, the respective programs typically have their own user interface, must be invoked with different command line parameters, and generally neglect interprocess communication with other applications. Furthermore, many tools are OS-dependent and extract pieces of evidence solely based on hard-coded offsets. Especially the latter aspect becomes a significant burden as the layout of important system structures, e.g., the `_EPROCESS` block, is frequently changed across different Windows versions and service pack levels. Thus, with new or updated operating systems, the respective solutions quickly become dated.

In order to cope with these issues, Walters and Petroni (2007) suggest integrating memory forensics-related investigate methods with the forensic process model illustrated by Carrier and Spafford (2003). In this model, a security incident leads to a *digital crime scene*, and an investigator is advised going through several individual phases to best preserve, secure, and finally extract and present relevant pieces of evidence. As Walters and Petroni (2007, p. 2) point out, such a model "allows us to organize the way we think about, discuss, and implement the procedures that are typically performed during an investigation" and "forces us to think about (...) how the tools and techniques that are used during a digital investigation fit together to support the process". Taking these ideas into consideration, several memory forensic *frameworks* were developed in the following years (Chan et al., 2009, 2010; Okolica and Peterson, 2010a). However, only the *Volatility Framework* has received broad attention in the forensic community to the best of our knowledge and is regarded as the de-facto standard memory analysis suite at the time of this writing (Volatile Systems, LLC, 2008, 2013a). In the following, we will give a brief overview of its major characteristics. A more thorough description of its mode of operation will be subject of Chapter 6.

One of the great benefits of the Volatility Framework is its modular architecture. In the default installation, investigators are already provided with a large number of individual modules for different purposes, e.g., for process- or file-related tasks.[8] However, the base functionality can also be conveniently extended when necessary through separate *plug-ins* that are automatically inserted into the module tree at program startup. Although plug-ins can theoretically work completely independently from other parts of the application (except from calling some base functions), many modules implement existing code and reprocess previous output and results. Due to this design hierarchy, development efforts are reduced to a minimum, and new features can be added to the framework within

[8] A short overview and description of existing modules for the Volatility Framework is available on the respective project homepage (Volatile Systems, LLC, 2013a).

a short time. At the time of this writing, Volatility supports most recent versions of the Microsoft Windows product family, both in the 32- and 64-bit variant, apart from Windows 8. The latest project release (2.3) permits examining other operating systems such as Linux, Mac OS, and Android as well. Platform-specific details about data structures and data types are thereby defined in the form of separate system *profiles*. Because these profiles rely on static offsets though, the analysis process can, under certain circumstances, be thwarted by simply modifying a single byte (see Haruyama and Suzuki, 2012). Practical countermeasures for such *anti-forensic* attacks are still greatly missing, thus, analysts should carefully scrutinize generated reports when investigating a snapshot of a suspicious system.

## 5.3 Summary

In this chapter, we have described techniques, strategies, and concepts for analyzing an image of a computer's RAM. Irrespectively of the artifact in question, two general methods for extracting evidence are either *logically* reconstructing resources as they were formerly maintained by the operating system or *physically* scanning the memory snapshot with the help of signatures. In the latter case, even objects whose life cycle has ended can be frequently found. By matching the two views, it is also possible to reveal little inconsistencies that potentially indicate the presence of a threat. We will see examples for such scenarios in Chapter 6.

As we have pointed out, many utilities for memory analysis-related tasks that were developed in the past were originally designed rather as a proof-of-concept demonstration than with a holistic forensic process in mind. Consequently, products were often only compatible with specific operating system versions and lacked interoperability with other applications. Due to these characteristics, their applicability in practice was limited. As a viable alternative, security professionals have therefore proposed the use of forensic frameworks that better take investigative process models into account. The *Volatility Framework* we have introduced in Section 5.2 has gained the widest attention in the forensic community. Because of its detailed analysis results, it is regarded as the standard solution for memory examinations to date. Case et al. (2008, p. S65) argue, however, that "merely swamping the user with all available data [...] falls well short of what is actually needed" and "is not, by itself, very useful". Thus, it is also important to develop suitable visualization techniques and properly present the collected evidence so that the completion of a case is not unnecessarily stalled. In the following chapter, we will illustrate how some of these issues can be addressed by automatically correlating individual findings and highlighting system areas that require particular attention.

# Chapter 6

# Using Memory Forensics to Discover Rootkit Infections

In the previous chapter, we have introduced the *Volatility Framework*, a powerful, free open source solution for memory analysis-related tasks (Volatile Systems, LLC, 2008, 2013a). Volatility is capable of efficiently processing RAM images of various system platforms and extracting a large number of forensic artifacts. While generated reports are rich in information though, they mainly aim at expert investigators who possess an in-depth knowledge of operating system internals. For less experienced practitioners, on the other hand, result interpretation is significantly more complex and demanding. The modular architecture of the framework becomes an additional burden in this case, because the output of executed modules must be manually matched and brought into connection. Especially when scanning a computer for malicious software, potential traces can therefore be easily overlooked. The latter argument is particularly true when dealing with *rootkits*, i.e., sophisticated malware species that attempt to thoroughly hide their presence on a machine and frequently only reveal themselves through subtle peculiarities during runtime.

In order to cope with these issues, we present the plug-in *rkfinder* for the popular *DFF* suite (ArxSys, 2009) in this chapter. DFF is a graphical investigation framework that has gained broader attention among security professionals after winning a forensic workshop challenge in 2010 (Jacob, 2010). Our plug-in integrates major functionality of the Volatility Framework into the DFF interface and creates an abstract, tree-like view of the system state. By automatically checking resources for consistency and highlighting possibly suspicious elements, potential sources of an incident can thus be identified and addressed more quickly, even by analysts who lack training in these areas. In contrast to existing rootkit and malware detection applications such as *IceSword* (Pan, 2005), *RootkitRevealer* (Cogswell and Russinovich, 2006), or *GMER* (GMER, 2013), our software does not need to be executed on a live system but can be launched in the course of a *post-mortem* examination. Thereby, forensic best practices such as the demand for *result reproducibility* or *impact limitation* can be taken into account more carefully (Mocas, 2004). Readers who would like to know more about the performance of the former technologies, however, are referred to the studies by Freiling and Schwittay (2007) and Todd et al. (2007).

### Outline of the Chapter

In the remainder of this chapter, we will focus on techniques and concepts for discovering rootkit infections. For this purpose, we first give a brief definition of the rootkit term in

Section 6.1, illustrate the main characteristics of these threats, and shortly describe the different classes rootkits are typically categorized into. In addition, we outline common system manipulation strategies and discuss how these manipulations can be revealed during a forensic memory analysis. In Section 6.2, we present our *rkfinder* plug-in for the DFF framework that facilitates some of the latter tasks. An evaluation of the developed software is subject of Section 6.3. In Section 6.4, we describe major weaknesses and limitations of the plug-in and suggest several possibilities for improving its capabilities and detection rate in the future. We conclude with a short summary of our work in Section 6.5.

## 6.1 Background Information

In the following section, we briefly recapitulate major properties of rootkits and introduce the prevailing rootkit classes that define the basic mode of operation of these threats. An understanding of the individual classes does not only help estimate the level of potential system modification that can be achieved by a particular species, but also indicates the level of effort that is required to detect its presence. An overview of fundamental rootkit techniques as well as a description of how these techniques can be revealed in the course of a memory-based analysis are subject of Section 6.1.2.

### 6.1.1 Rootkits and Rootkit Classes

According to Hoglund and Butler (2005, p. 4), a rootkit "is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer". Taking this definition into consideration, major capabilities of such a program are thus hiding specific system resources, e.g., files, network connections, or registry keys, from legitimate system users and, preferably, other, security-related applications. In sum, the main purpose of a rootkit is helping an attacker *maintain access* once a computer has been compromised.

Rootkits are typically categorized into different classes, depending on the system layer they operate on (Farmer and Venema, 2005a; Carvey, 2007; Davis et al., 2010):

- *Application-level* or *user-space* rootkits run in non-privileged mode on *ring 3* of the x86 processor architecture, i.e., they have limited access to hardware resources. In the past, this type of malicious software simply replaced or modified trusted (user-space) binaries with malicious content. In contrast, a common technique on Microsoft Windows operating systems to date is to intercept specific function calls, e.g., to the `FindFirstFile` function for searching files, in order to manipulate data processing or filter results. Because these operations are performed on a high system layer though, application-level rootkits are comparatively the easiest to detect, particularly by programs being rooted deeper in the system core. A typical representative of this species is the widely spread *Hacker Defender* (HolyFather, 2005).

- *Library-level* rootkits, e.g., *NTIllusion* (Kdm, 2004) or *Vanquish* (XShadow, 2005), frequently run on ring 3 of the x86 processor architecture as well. As opposed to application-level rootkits, they solely target specific system components though, e.g., the standard user-mode library `kernel32.dll`. As the code of these files is injected into every process, a large number of programs can be compromised at the same time with only little effort.
- *Kernel-level* rootkits such as *FU* (Butler, 2005) or *FUTo* (Silberman and C.H.A.O.S., 2006) typically comprise a malicious device driver and aim at the lowest layer of the operating system. Once in kernel mode, system structures can be substantially modified, because the respective code is executed in privileged mode on *ring 0* of the x86 processor architecture with full access permissions. Due to these characteristics, this type of threat can severely affect the integrity of a machine. On the other hand, kernel-level rootkits are usually tightly bound to a platform and must be carefully designed because even the slightest programming error may lead to a system crash that potentially raises suspicion.

Please note that two other classes of rootkits, *virtualized* and *firmware-based* rootkits, are frequently distinguished in the literature as well (Hoglund and Butler, 2005). As these types of threats are not in the focus of our work and potentially require special hardware for detection (Petroni et al., 2004), they are not further considered in the scope of this thesis though.

### 6.1.2 Common Rootkit Strategies and Techniques

Rootkits may choose from a variety of system manipulation techniques in order to present a modified view of the computer's state to the user. In the previous chapter, we have already described *Direct Kernel Object Manipulation* (DKOM) and malicious *library injection* attacks. In the former case, core operating system structures are directly changed in memory. A typical example we have illustrated is unlinking specific `_EPROCESS` objects from the internal `ActiveProcessLinks` list. Thereby, individual processes can be effectively hidden from common system administration programs such as the *Task Manager*. In the case of a library injection attack, on the other hand, a malicious DLL is inserted into the address space of a legitimate application, e.g., by calling the `LoadLibrary` API function. Because the list of loaded libraries is maintained in a substructure of the user-level *Process Environment Block* (PEB, see Section 5.1.5), the respective information is particularly easy to subvert. Rootkits can employ several other system compromising strategies as well though. In the following sections, we will give a brief overview of the most prominent ones and discuss how these can be discovered during memory analysis.

#### Hooking

One of the most widely adopted mechanisms for observing or adapting the program flow of an application is installing a so-called *hook* on an infected machine. A hook

intercepts calls to certain system functions and temporarily transfers execution to a custom routine, the *hooking function*, that is under control of an attacker. With the help of the hooking function, parameters and return values can then be modified in the next step. For instance, the *Hacker Defender* rootkit monitors calls to the Windows API functions `EnumServicesStatus` and `EnumServicesStatusEx` to prevent pre-defined system services from showing up in the service list (HolyFather, 2005). For this purpose, the prologue of the respective functions is overwritten *inline* (directly in memory) with an unconditional jump to the hooking procedure. By calling a special *trampoline* function, the original instructions can be restored later, and the remainder of the original function code is run.

In a forensic investigation, the previous operations are trivial to detect by disassembling the prologue of the functions in question and checking them for relative jumps to an external resource. With a similar approach, less common hooking techniques that target the import (IAT) or export address table (EAT) of an executable can be revealed as well. More details about this process can be found in the work of Ligh et al. (2010).

#### Manipulation of the Service Control Manager

Kernel-level rootkits frequently load their malicious driver into the system core by temporarily setting up a rogue system service that is executed with elevated privileges. When a service is created, an entry to the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Service` key in the Windows registry is added by default though. In order to cover their traces, rootkits therefore commonly delete said entry once the service has been started. As a side effect of this operation, the service cannot be stopped via system administration programs such as the *Microsoft Management Console* (MMC) anymore either (Ligh et al., 2010). However, it can still be easily enumerated by running a simple search query over the command line interface (I.M.Weasel, 2006).

To cope with the latter issue, more sophisticated species additionally modify the internal service database. This database is maintained by the *Service Control Manager* (SCM, `services.exe`) and consists of objects of type `_SERVICE_RECORD` that are connected with each other through a doubly-linked list (see offset `0x000` in Figure 6.1). Thus, to properly hide a service, it is sufficient to remove the respective record from the list, similarly to the process manipulation attack described in Section 5.1.1. Nonetheless, the structure itself is still physically kept in memory and, consequently, can be found by scanning the address space of the `services.exe` instance. For this purpose, a unique signature needs to be generated. As each service record includes a *tag* field with the constant value `sErv` at offset `0x018`, this is a trivial task though.

## 6.2 Finding Traces of System Infections with *rkfinder*

To facilitate the identification of potentially suspicious elements in the course of a memory-based analysis, we have designed *rkfinder*, a plug-in for the popular *Digital*

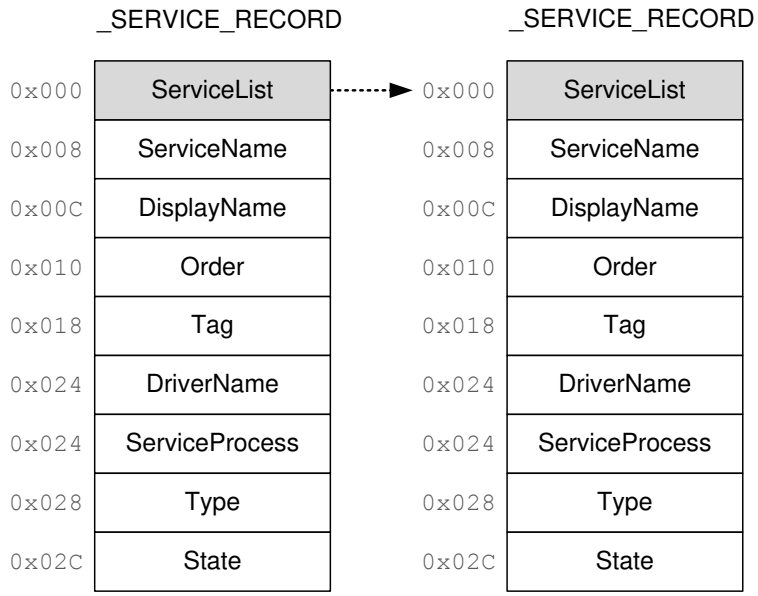



Figure 6.1: Internal Structure of a System Service
(Based on Ligh et al., 2010, p. 663)

*Forensics Framework* (DFF) suite (ArxSys, 2009). As we have already indicated at the beginning of this chapter, DFF is a free, open source investigative platform and has gained a good reputation among researchers and practitioners alike after winning a forensic challenge in 2010 (Jacob, 2010). In comparison to similar products such as *PyFlag* (Cohen, 2008), DFF has a "clean, simple GUI" and can be efficiently operated even by novice investigators (Altheide and Carvey, 2011, pp. 221-222). Furthermore, the framework offers a modular architecture and, thus, can be easily extended when needed. Our rkfinder plug-in makes use of this capability and integrates various components of the Volatility Framework into the graphical DFF interface. Thereby, analysts are able to comfortably examine memory snapshots of a suspicious system and get a comprehensive overview of the system state within a short time.

In the following section, we illustrate the functionality of DFF in more detail and explain how our plug-in is integrated into the framework. A description of rkfinder's mode of operation is subject of Section 6.2.2.

### 6.2.1 Functionality and Extension of DFF

The DFF platform is internally divided into three layers (ArxSys, 2012): The *core* layer provides fundamental functions that are required to execute the different software modules and process their respective output. Part of this layer is the *Virtual File System* (VFS), a hierarchy that is responsible for organizing the analyzed data. Information

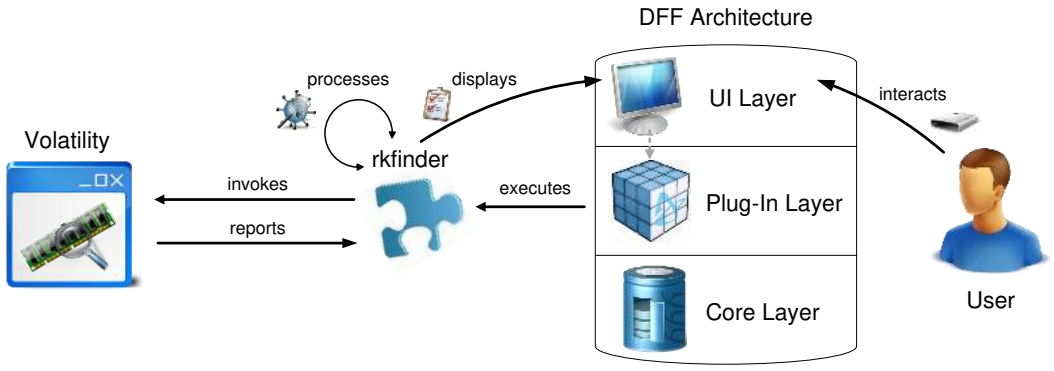



Figure 6.2: Integration of the *rkfinder* Plug-In in the DFF Architecture

that is associated with the VFS is represented in the form of *nodes*. By default, a node consists of a `name` and `size` attribute, but can be extended to include further details. For instance, the rkfinder plug-in defines a *process* node with various attributes that store the name, identifier as well as start and end times of an executable.

Tightly linked with the core is the *plug-in* layer that allows extending the framework with specific features. For this purpose, individual modules need to be written, either in the Python or C++ programming language. Modules inherit basic node manipulation methods, e.g., for reading or changing specific values, from a parent class and can be easily "stacked" to create different levels of abstraction. For example, the rkfinder plug-in is capable of automatically dumping portions of memory that may contain malicious shellcode. With the help of a second module, e.g., `hexedit`, a user can then examine the extracted memory regions in more detail if desired. Results are finally presented on the *user interface* layer, i.e., the graphical component of the framework. The user interface permits structuring, aggregating as well as filtering extracted pieces of evidence and, thus, explicitly supports the *documentation* and *presentation* phase described in classic forensic process models (Casey, 2011).

Our rkfinder plug-in is designed as a separate module that mainly operates on the plug-in and user interface layer (see Figure 6.2). Therefore, it is able to run independently from other parts of the DFF software. In particular, as the heart of the platform is left untouched, the plug-in can be executed on any operating system the original code can run on, i.e., both on versions of the Microsoft Windows as well as Linux product family. The module-specific approach of the plug-in also ensures quick installation and future maintenance, because the respective files only need to be copied to the DFF working directory. On the other hand, the internal functionality of rkfinder is largely encapsulated. Thus, after loading a memory image, the investigation process may be started with a single click, and a summary report is automatically generated. As rkfinder especially targets practitioners with lesser developed forensic expertise, these characteristics are of great benefit. A more in-depth description of its mode of operation is subject of the next section.

### 6.2.2 Rkfinder's Mode of Operation

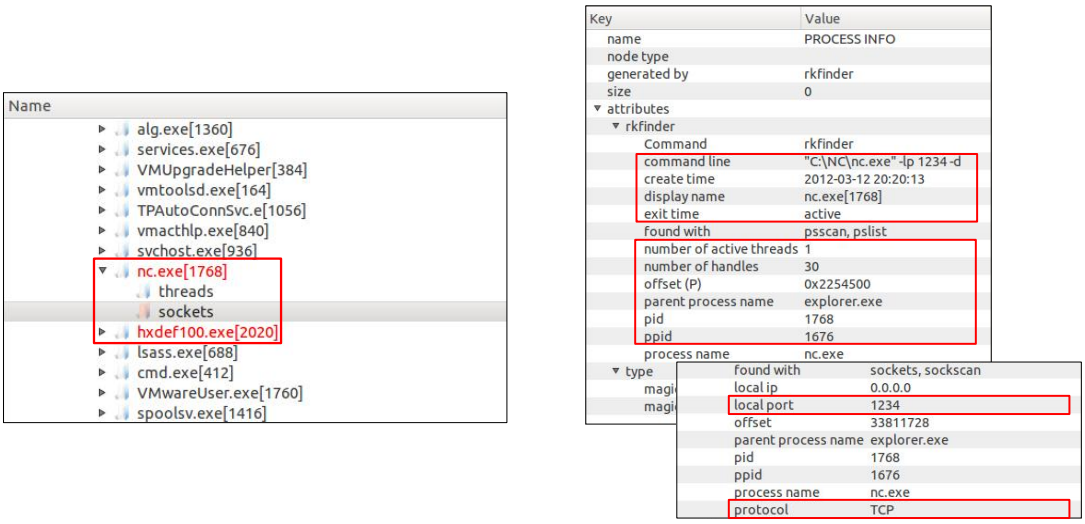
Rkfinder implements *cross viewing* techniques for rootkit detection, i.e., possibly suspicious elements are discovered by examining the computer's state from different angles and checking results for inconsistencies. The approach is based on the assumption that anomalies that are identified in this process frequently indicate the presence of a threat, because it is difficult for malicious software to consistently subvert operating system structures and evade *both* post-mortem and live investigative measures. Internally, rkfinder invokes various modules of the *Volatility Framework* in the background and automatically associates extracted artifacts. For this purpose, the object hierarchy of the application is imported into DFF in a first step. By hooking the `calculate` method, i.e., a generic helper function that must be implemented by each module and is responsible for executing module-specific tasks, we are then able to monitor memory analysis-related operations and obtain the respective module output for further investigation within DFF.

The modules that are required for detecting the common rootkit deception strategies we have described in the previous chapter and Section 6.1.2 are summarized in Table 6.1. As can be seen, indicators for system infections are discovered by both restoring logical system structures and physically scanning the image file. A comparison of these layers leads to the first *cross view* we have implemented in rkfinder. For instance, to generate the entire list of running processes on the machine, we match the output of the `pslist` and `pslinks` modules. While the former one helps recover the elements of the `ActiveProcessLinks` list, the latter module searches a memory snapshot for instances of `_EPROCESS` objects. As we have explained in Section 5.1.1, any discrepancy between these two sources strongly suggests a *Direct Kernel Object Manipulation* (DKOM) attack. Such methods are actively employed by kernel-level rootkits, e.g., *FU* (Butler, 2005) or *FUTo* (Silberman and C.H.A.O.S., 2006). One pitfall when parsing results, however, is that data are frequently returned inconsistently and, therefore, need to be carefully validated in order to avoid false positives. Likewise, although certain rootkit techniques such as user-space *hooks* can be revealed quite comfortably (see Section 6.1.2), the corresponding hidden resources may be more difficult to detect and can be easily overlooked unless they are appropriately highlighted. For example, the *Hacker Defender* rootkit intercepts various Windows API functions to conceal pre-defined elements from system maintenance programs at runtime (HolyFather, 2005). In a post-mortem analysis, these measures are no longer effective though. Consequently, the respective objects are regularly displayed in forensic applications and, thus, are harder to distinguish from legitimate artifacts, especially if an attacker tries to obscure (file) names and attributes. In this case, finding all traces of an incident can be a cumbersome and challenging task, particularly for less experienced practitioners.

In order to cope with the latter issue, rkfinder is capable of considering reports of a basic *live response* in its analysis. These reports can include information gained with standard process or network administration tools, e.g., the *Task Manager* or the built-in *netstat* utility. Ideally, the programs are run shortly after the snapshot of the computer's RAM has been acquired. Thereby, data pollution due to concurrent activity is reduced

| Module Name | Description |
|---|---|
| `apihooks` | Searches the memory image for possibly installed malicious Windows API hooks (see Section 6.1.2). |
| `connections` | Enumerates the list of open network connections by locating the `tcpip.sys` driver file in memory and iterating the `_TCPT_OBJECT` list (see Section 5.1.4). |
| `connscan` | Physically scans the snapshot with the help of signatures for `_TCPT_OBJECT` structures to find open network connections (see Section 5.1.4). |
| `dlllist` | Obtains the list of loaded libraries by analyzing the list of `LDR_DATA_TABLE_ENTRY` structures that are referenced in the Process Environment Block of a process (see Section 5.1.5). |
| `filescan` | Physically scans the memory image with the help of signatures for `_FILE_OBJECT` structures to find open files (see Section 5.1.5). |
| `malfind` | Parses the Virtual Address Descriptor (VAD) tree of a process to identify memory pages that possibly contain malicious shellcode (see Section 5.1.5). |
| `modscan` | Physically scans the memory dump with the help of signatures for `LDR_DATA_TABLE_ENTRY` structures to find loaded (kernel) modules (see Section 5.1.5). |
| `modules` | Obtains the list of loaded kernel drivers by analyzing the list of `LDR_DATA_TABLE_ENTRY` structures, starting from the global symbol `PsLoadedModuleList` (see Section 5.1.5). |
| `pslist` | Iterates members of the `ActiveProcessLinks` to enumerate the list of running processes (see Section 5.1.1). |
| `psscan` | Physically scans the memory image for `_EPROCESS` structures to generate the list of (currently and previously) running processes (see Section 5.1.1). |
| `sockets` | Enumerates the list of open sockets by locating the `tcpip.sys` driver file in memory and iterating the `_ADDRESS_OBJECT` list (see Section 5.1.4). |
| `sockscan` | Physically scans the memory image with the help of signatures for `_TCPT_OBJECT` structures to find open sockets (see Section 5.1.4). |
| `svcscan` | Physically scans the address space of the Service Control Manager with the help of signatures for `_SERVICE_RECORD` structures to find hidden services (see Section 6.1.2). |
| `thrdscan` | Physically scans the memory image for `_ETHREAD` structures to create the list of (currently and previously) running threads (see Section 5.1.1). |
| `threads` | Applies various heuristics, e.g., matching the `ExitTime` field with other fields of an `_ETHREAD` structure, to determine the list of running threads (see Ligh, 2011a). |
| `vadinfo` | Obtains information about a node in the Virtual Address Descriptor (VAD) tree, including the VAD flags that, for instance, define the access permissions for the respective pages (see Section 5.1.5). |

Table 6.1: List of Volatility Framework Modules that are Integrated in *rkfinder*




(a) System Process View with Automatic Highlighting of Suspicious Elements

(b) Meta Information View with Detailed Information about a Selected Item

Figure 6.3: Visual Analysis of Threats with *rkfinder*

to a minimum, and the risk of introducing false positives is decreased. As we will see in Section 6.3, by matching the results of a live analysis with a post-mortem investigation, a *second* cross view layer is created that helps discover many common rootkit species found "in the wild" today.

Once all operations are completed, findings are summarized in a tree-like pane of the graphical DFF interface. In the left pane, different categories of objects can be chosen. For example, in Figure 6.3a, the extracted *process* hierarchy of an examined system is shown. When clicking on a node, users can obtain further information about the threads, loaded libraries, and open network connections that are associated with an application. Additional meta information about an executable are displayed in the right pane of the interface, e.g., its command line, the start and end time, the number of references (handles) to files and other resources, as well as the name and identifier of the parent structure. Items that are of particular interest or deserve closer attention are automatically highlighted. In the depicted example, rkfinder detected inconsistencies for the two processes `nc.exe` and `hxdef100.exe`. These processes are automatically marked in red and, thus, immediately make a user aware of a potential problem. Clicking on the first item reveals that a (hidden) socket was opened on port `1234` of the system (see Figure 6.3b). In contrast, the second process represents the main component of the *Hacker Defender* rootkit that is concealed under normal circumstances at runtime, but has been successfully discovered by rkfinder with the help of cross viewing techniques. Based on these results, an investigator can subsequently analyze, for instance, the individual threads that were started by the two programs or request a report about the memory regions malicious instructions were inserted into.

Apart from the process-specific view, rkfinder also offers a *network*, *services*, and *system library* perspective that allows quick browsing through pieces of evidence. Thereby, users are able to gain a comprehensive overview of the system state within a short time. Because all suspicious objects are finally listed in a separate category, even investigators with lesser developed forensic expertise are able to comfortably examine and address critical system areas that have potentially been infected by a threat. In comparison to the original, command line-based unrelated output of the Volatility Framework, this is a major benefit.

## 6.3 Evaluation and Discussion of the Detection Performance

We have evaluated our software in a preliminary study with the help of different memory images acquired from six rootkit-infected machines. All machines were running a default installation of Microsoft Windows XP (SP 2) and were set up within a virtual (*VMware*-based) environment. This configuration permitted suspending a system at any time and easily duplicating the created RAM snapshot for later analysis (see our explanations regarding virtual machines in Section 2.2.3). To simulate a realistic attack, we copied a number of security- and network-related applications to each system. For instance, we installed *netcat* (Giacobbi, 2006), a simple utility for sending and receiving data over a network connection, and created a basic listening service on a local port. The rootkits were configured to hide all runtime information and resources involved, provided the respective capability was offered by the malicious software.

Once a system had been infected, we temporarily stopped its execution process and obtained a snapshot of its physical memory. Immediately after the acquisition phase, we briefly resumed the machine and launched a custom script that invoked certain operating system commands to determine, for example, the list of running processes, open network connections, and installed services. The output of these commands was saved to a file and, together with the previously generated memory image, transferred to a trusted workstation. The image was then opened in *DFF*, and the *rkfinder* plug-in was started. As described in Section 6.2, rkfinder correlated the output of various modules of the *Volatility Framework* and matched the external, post-mortem analysis view with the internal view of the operating system. Thereby, system inconsistencies could be effectively revealed. Primary objective of our evaluation was assessing in how far rkfinder – in cooperation with the Volatility Framework – was capable of identifying and properly visualizing all rootkit-related manipulations. Precisely, a rootkit was considered as being *discovered* when the respective system modifications and affected artifacts were correctly highlighted in the DFF interface. As we have argued in the previous section, we believe that indicating the sources of a potential system infection is particularly helpful for inexperienced practitioners. A detailed study of our software with respect to this target group is, however, still missing at the time of this writing.

### 6.3.1 Analysis Results

In the following, we present the results of our analysis. In Table 6.2, the list of considered rootkits is shown. The different species represent examples of the rootkit classes we introduced in Section 6.1.1 and are commonly found "in the wild" to date (Davis et al., 2010). As can be seen, all threats are able to hide at least individual processes. Most rootkits, however, implement additional capabilities such as concealing network connections, loaded drivers, or entries in the system registry. In the course of a forensic analysis, these manipulations should be revealed to a preferably high degree. The corresponding detection results with respect to our plug-in are listed in Table 6.3. A checkmark (✓) denotes that a (hidden) resource could be successfully discovered. In contrast, a dash (-) denotes that the system modification remained unnoticed. In case a rootkit was incapable of hiding a specific object, the entry in the respective column was labeled as `n/a`.

We now discuss the results shown in Table 6.3 more thoroughly: Generally speaking, rkfinder could successfully identify all modifications apart from maliciously inserted registry keys. The latter functionality, although planned, is not included in our software yet, because it is possible that portions of the registry are either paged out to disk or have not been transferred into memory at a given point in time (see our explanations in Section 5.1.3). For this reason, more testing is required from our part to verify the reliability of this feature. Furthermore, even though *FU*, *FUTo*, and *Hacker Defender* were assumed to effectively hide system drivers according to their documentation, these operations failed for all three rootkits, and the items were regularly displayed in system maintenance programs and, thus, in the module list of our plug-in as well. Because the external and internal view did not yield any differences, however, rkfinder was not able to detect any anomalies. Therefore, the respective items could not be highlighted as being suspicious. The same is true for *NTIllusion* that did neither succeed at concealing the `netcat` process from standard task management programs nor removing the injected library from the corresponding doubly-linked lists in the Process Environment Block (see Section 5.1.5). Again, due to the missing mismatch between the external and internal perspective, these elements were not marked as suspicious even though they were included in rkfinder's list of running processes and loaded libraries. In spite of this, it is noteworthy that other authors (Freiling and Schwittay, 2007; Dolan-Gavitt, 2007a) have reported NTIllusion to work as expected. Further testing, e.g., with different versions of the software, is therefore required to properly assess the behavior of the threat.

Concerning the system manipulation techniques outlined in Chapter 5 and Section 6.1.2, it is interesting to verify if and to what degree the individual rootkit methods could be discovered: *Direct Kernel Object Manipulation* was implemented in two kernel-level species, *FU* and *FUTo*. These operations could be revealed by traversing the corresponding linked lists of objects and comparing the individual elements with the results of signature-based scanners. For this purpose, rkfinder correlated the output of several Volatility modules, e.g., `pslist` and `psscan` (see Section 6.2.2), and successfully detected the presence of both rootkits. In contrast, marking the *maliciously injected library* of

| **Rootkit** | **Type** | **Supports Process Hiding** | **Supports Registry Key Hiding** | **Supports Socket Hiding** | **Supports Service Hiding** | **Supports Driver Hiding** |
|---|---|---|---|---|---|---|
| BH-Rootkit-NT | *K* | ✓ | - | ✓ | - | - |
| FU | *K* | ✓ | - | - | - | ✓ |
| FUTo | *K* | ✓ | - | - | - | ✓ |
| Hacker Defender | *U* | ✓ | ✓ | ✓ | ✓ | ✓ |
| NTIllusion | *L* | ✓ | ✓ | ✓ | - | - |
| Vanquish | *L* | ✓ | ✓ | - | ✓ | - |

Table 6.2: List of Evaluated Rootkits

(*Note:* A rootkit of type *K* refers to a *kernel-level rootkit*, a rootkit of type *U* to a *user-level rootkit*, and a rootkit of type *L* to a *library-level rootkit*.)

| **Rootkit** | **Process Detection** | **Registry Key Detection** | **Socket Detection** | **Service Detection** | **Driver Detection** |
|---|---|---|---|---|---|
| BH-Rootkit-NT | ✓ | n/a | ✓ | n/a | n/a |
| FU | ✓ | n/a | n/a | n/a | - |
| FUTo | ✓ | n/a | n/a | n/a | - |
| Hacker Defender | ✓ | - | ✓ | ✓ | - |
| NTIllusion | - | - | ✓ | n/a | n/a |
| Vanquish | ✓ | - | n/a | ✓ | n/a |

Table 6.3: Detection of Rootkit-Manipulated Objects

the *NTIllusion* rootkit as suspicious failed, because a comparison of the loaded module list with resources defined in the Virtual Address Descriptor tree (see Section 5.1.5) did not reveal any inconsistencies. With respect to the second library-level rootkit *Vanquish*, this procedure proved fruitful, however. Rkfinder also succeeded in identifying the malicious code parts the two threats injected into memory pages.

Two rootkits, *Hacker Defender* and *Vanquish*, are known to install custom *services* that are hidden from the Service Control Manager. With the help of the `svcscan` module that is part of the *Volatility Framework*, these modifications were correctly recognized, and the corresponding services were discovered. Last but not least, *hooking* attempts were actively pursued by four out of six rootkits, i.e., *Hacker Defender*, *NTIllusion*, *Vanquish*, and the kernel-based *BH-Rootkit-NT*. In our evaluation, rkfinder was capable of finding all hooks that were mentioned in the respective manuals. As we will point out in Section 6.4.1, however, a hooked function does not necessarily indicate the presence of a threat. For this reason, investigators should examine the respective list with care. A final summary of the individual detection rates is given in Table 6.4.

| Rootkit Technique | Employed by | Detection Rate |
|---|---|---|
| DKOM | FU, FUTo | 2/2 |
| Library Injection,<br>Code Injection | NTIllusion, Vanquish | 1/2<br>2/2 |
| Service Manipulation | Hacker Defender, Vanquish | 2/2 |
| Hooking | BH-Rootkit-NT, Hacker Defender, NTIllusion, Vanquish | 4/4 |

Table 6.4: Detection Rates for Different Rootkit Techniques

## 6.4 Discussion

In the following, we briefly depict major weaknesses and limitations of rkfinder that should be taken into consideration when running the plug-in in practice. In Section 6.4.2, we discuss several possibilities for further extending the capabilities of the software and improving its detection performance in the future.

### 6.4.1 Weaknesses and Limitations

In the development process of the rkfinder plug-in, strong emphasis was put on correctly highlighting system modifications as well as suspicious elements, while reducing the number of false positives to a minimum. Although we were capable of successfully detecting common rootkits, the software still has to struggle with several weaknesses and limitations that have to be better addressed in the future: First of all, as we have indicated in the previous section, hooked functions that are found on a machine are not necessarily a sign of a system infection but may be installed by legitimate, security-related applications. In fact, Freiling and Schwittay (2007, p. 15) argue that "identifying malicious hooks can sometimes be difficult, because a lot of software, especially Antivirus software, uses benign hooking".

Moreover, even though we managed to discover the injected library of the *Vanquish* rootkit by parsing the Virtual Address Descriptor tree, Dolan-Gavitt (2007a) has pointed out that the structure is susceptible to Direct Kernel Object Manipulation. Such *anti-forensic* techniques may significantly affect the progress of an investigation and falsify results. Oddly enough, however, malicious applications that do not actively cover their traces are the hardest species to mark as suspicious in the end, because the cross view approach does not lead to any inconsistencies. In this case, an investigator must manually analyze the reports of the plug-in and identify sources of a system infection without additional program support.

Last but not least, it is important to keep in mind that, similarly to other detection programs, the rkfinder software can reflect a deceptive picture of a computer's state in case the presence of a threat remains unnoticed. From a psychological point of view, these *false negatives* are especially dangerous, because users may incorrectly perceive the

level of system security as satisfactory and, as a result, possibly react less sensitively to future anomalies. Even though our methodology of matching the internal and external view has proven capable of identifying common rootkits present "in the wild" to date, we have also mentioned other classes of malware, for instance, virtualized rootkits such as *Blue Pill* (Rutkowska, 2006), that would be impossible to reveal with our plug-in. In such a situation, consulting more experienced forensic analysts is advisable.

### 6.4.2 Further Development and Evaluation Possibilities

So far, the performance of rkfinder has unfortunately only been assessed with a limited number of rootkits. Even though the considered samples are frequently involved in attacks on Microsoft Windows platforms (Carvey, 2007; Davis et al., 2010), they are mostly dated, and their behavior is well documented in the literature. For this reason, the study should be significantly extended, and more modern and sophisticated species should be included in the evaluation. Researchers have shown that renowned threats such as *Stuxnet* (Falliere et al., 2011) or *Flame* (Laboratory of Cryptography and System Security, 2012) can be successfully discovered with the help of the Volatility Framework (Ligh, 2011b, 2012), and it is expected that these capabilities can be integrated into our plug-in as well. In fact, by adopting malware classification mechanisms as they are offered by solutions such as *Yara* (Alvarez and Wiacek, 2013), it would be possible to examine other types of malicious software as well, e.g., trojan horses or backdoors. Thereby, rkfinder could evolve to a more generic solution.

With respect to the development process of the plug-in, it is necessary to implement additional detection features. In particular, the software should be able to parse in-memory registry data, e.g., the well-known *run* keys such as `HKLM\Microsoft\Windows\CurrentVersion\Run` that are commonly manipulated by malware to automatically load applications at system start (see Carvey, 2011). Furthermore, discovery rates may be increased by better taking certain heuristics into account. For instance, Ligh et al. (2010) have proposed analyzing the execution priority of threads to distinguish suspicious behavior from legitimate system activity. Other suggestions describe verifying the parent-child hierarchy of processes or the list of access privileges. On the other hand, these types of consistency checks greatly bear the risk of introducing new false positives. For this reason, the respective program components should be carefully tested before being used in a forensic investigation.

## 6.5 Summary

In this chapter, we have presented *rkfinder*, a plug-in for the popular open source investigation platform *DFF*. Rkfinder integrates major functionality of the *Volatility Framework* into the graphical DFF interface. By processing, correlating, and filtering the respective module output, the plug-in is able to discover and automatically highlight potential traces of a system compromise, given a memory snapshot of the machine in question.

Thereby, analysts can quickly get a comprehensive overview of the system state and gain a solid foundation for addressing a threat in the course of a first response. Due to these characteristics, rkfinder particularly aims at users with solely little forensic expertise.

In a preliminary study, we have evaluated the software based on six rootkit samples that are frequently faced in practice to date. The rootkits operated on different operating system layers and were capable of manipulating system structures in both user as well as kernel space. With the help of several *cross viewing* techniques, we could reveal these manipulations to a high degree. However, weaknesses of the software are still its inability to successfully parse the Windows registry and correctly distinguish malicious from legitimate function hooks as, for instance, they are installed by diverse security applications. By addressing these issues in later versions of rkfinder, the detection quality of the plug-in can be further improved, and post-mortem memory investigations can be performed even more thoroughly.

# Chapter 7

# Synopsis and Conclusion

As we have seen in the previous chapters, a myriad of valuable information are contained in volatile memory that may significantly help an investigator complete a case. The different artifacts are extremely fragile, however, and can be easily destroyed or overwritten by cutting power to the machine or executing commands on the system. To avoid unintentional data loss, particular attention must therefore be paid to properly securing pieces of evidence in the first step. In Chapter 2 of this thesis, we have given an overview of available technologies for these tasks. Existing solutions include, for instance, special PCI cards that are capable of creating a snapshot of a computer's RAM when an external switch is activated. Since those cards must be installed and set up prior to their use though, they are only practical in controlled environments. As a viable alternative, we have illustrated possibilities for duplicating the address space via hardware buses, e.g., the IEEE 1394 (*FireWire*) interface. The respective techniques are easy to apply even in incident response situations, yet they are also susceptible to manipulation as a publication by Rutkowska (2007) has shown. Similar concerns hold true for software-based imagers that rely on functions provided by the target operating system. Because imaging utilities must be executed in parallel to other applications, produced memory snapshots are typically not completely consistent either. On the other hand, other proposals such as injecting a second, miniature OS into the kernel or performing a *cold boot* attack on the host suffer from certain limitations and drawbacks as well. In sum, none of the existing approaches we have described is immaculate. For this reason, analysts need to carefully balance the characteristics and benefits of the individual methods with regard to a given scenario.

In order to assess acquisition technologies more thoroughly, we have introduced and formalized three criteria that determine the quality of a forensic memory snapshot in Chapter 3. Specifically, a snapshot is defined to be *correct* with respect to a set of addressable memory regions if it solely contains the "true" values of these regions at the time the snapshot was taken. In contrast, a snapshot is said to be *atomic* if it is free of the signs of concurrent activity. By the level of *integrity*, we refer to the stability of values stored in memory during the time of the imaging period. With the help of the three criteria, we have evaluated the performance of selected acquisition solutions in Chapter 4. For this purpose, we have developed an extensive testing platform that, for the first time, allows an in-depth and repeatable examination of imaging utilities. Our platform is built upon the *Bochs* x86 PC emulator (The Bochs Project, 2013a) and generates a detailed protocol of all relevant operations. In a preliminary study, we have analyzed a total of 270 RAM images and have shown that two out of three considered evaluation candidates were initially unable to copy the physical address space in its entirety. We have fixed

the affected program components so that correct snapshots were eventually produced. We could also prove that even on idle systems, software-based imagers create forensic snapshots with a significant degree of inconsistency. However, it is an open research problem at the time of this writing whether or not these inconsistencies truly have an impact on later phases of an investigation.

In Chapter 5 we described procedures for analyzing forensic images. Precisely, we explained methods, concepts, and strategies for reconstructing the list of processes, open network connections, and referenced files. In addition, we discussed approaches for extracting cryptographic keys and remnants of the Microsoft Windows registry. As we have pointed out, artifacts can be successfully recovered in many cases by either restoring a *logical* view on resources, i.e., as they were originally seen by the operating system, or by *physically* scanning the memory dump for object structures. A comparison of the logical and physical perspective in a second step even permits identifying potential discrepancies in the system state that may indicate a malware infection.

Common system manipulation techniques as they are typically implemented in more sophisticated species such as *rootkits* were illustrated in Chapter 6. Examples included *Direct Kernel Object Manipulation* (DKOM), malicious *library injection*, and function call *hooking* in order to change the execution path of an application or filter results. We also introduced *rkfinder*, a plug-in for the popular forensic framework *DFF* (ArxSys, 2009), that integrates major functionality of the *Volatility Framework* (Volatile Systems, LLC, 2008, 2013a) into a comfortable graphical interface. Rkfinder employs various *cross viewing* algorithms for checking the integrity of a machine from different angles. Because suspicious elements are automatically highlighted, even users without expert knowledge in malware analysis are able to quickly recognize and address system areas that require particular attention. In sum, with the material presented in this thesis, practitioners have a solid basis for efficiently acquiring and examining volatile information in the course of a forensic investigation. For novice academic researchers, on the other hand, our insights may serve as a starting point for their own works. An overview of possible future research directions is subject of the following section.

## 7.1 Opportunities for Future Research

### 7.1.1 Anti and Anti-Anti Memory Forensics

As our evaluation of software-based imagers has shown in Chapter 4, acquisition solutions may contain logic errors and are therefore not capable of correctly duplicating the physical address space in all cases. With investigators more and more realizing the value of volatile information, however, attempts to actively stall or even block the imaging process in the first place will increase on the side of cyber criminals as well. Simple *anti-forensic* measures may include, for instance, controlling access to the `\\.\Device\PhysicalMemory` section object as outlined by Bilby (2006). In a recent publication, Stüttgen and Cohen (2013) have demonstrated that most imaging utilities available on the market to

date can be easily compromised by manipulating the return values of undocumented API functions the respective programs internally rely on. Even worse, because function calls are not adequately checked, some applications crash and, thereby, destabilize the entire system. Stüttgen and Cohen (2013, p. 2) conclude, "when facing determined and skilled adversaries, (...) the current generation of forensic memory acquisition tools are ill-equipped".

Taking these aspects into consideration, it is necessary to study the weaknesses of existing products in more detail and identify suitable defense mechanisms for possible attacks in a second step. For instance, with their miniature hypervisors that transparently transform an operating system into a virtualized guest, Martignoni et al. (2010) and Yu et al. (2012) have indicated possibilities for successfully coping with kernel-level threats in the course of the imaging period. In contrast, standard memory *analysis* frameworks can still be defeated comparatively trivially (see Haruyama and Suzuki, 2012). We believe that trying to improve the resilience of current technologies is an area that is well worth exploring.

### 7.1.2 Memory Forensics on Other System Platforms

In the scope of this thesis, we primarily described memory acquisition and analysis on machines running *Microsoft Windows*. In fact, the product family of Windows operating systems has been in the sole focus of researchers for a long time, while the intricacies of other platforms were utterly neglected. However, with the growing market share of Apple computers (Net Applications, 2013), investigators are likely to face an increasing number of *Mac OS*-based systems. Likewise, *Linux* operating systems and their derivatives are still the premier choice within server environments (E-Soft Inc., 2009). Data structures that are important for memory forensics-related activities on these systems have only been partially identified yet and still need to be examined in more detail. The same holds true for *Android* and *iOS* systems that are usually pre-installed on mobile phones or tablet PCs. Especially the former have caught broad interest among security professionals more recently, yet existing procedures mainly illustrate a collection of best practices and frequently only refer to specific product vendors or versions (Vidas et al., 2011). Establishing a more scientific methodology for the investigation of these devices will therefore be an interesting field of research in the future. A good overview of the current state of the art can be found in the works of Hoog (2011) and Sylve et al. (2012).

### 7.1.3 Virtual Machine Introspection

As part of a forensic memory analysis, raw data of the snapshot must be transformed into a structured and meaningful format. For example, in order to restore the process list, it is first necessary to locate the `PsActiveProcessHead` symbol in memory and subsequently enumerate the members of the `ActiveProcessLinks` list (see Section 5.1.1). For this purpose, the original virtual address space of the operating system has to be rebuilt.

Once the latter operations have been completed, even more information about the system state can be extracted. The process of deriving semantic knowledge from low-level entities is known as "bridging the semantic gap" (Chen and Noble, 2001) and is highly relevant for other digital forensic disciplines as well. One niche that has gained particular attention in the last years is *Virtual Machine Introspection* (VMI), i.e., "inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it" (Garfinkel and Rosenblum, 2003, p. 2). As we have argued in Chapter 2, with the increasing use of Internet-hosted services, securing evidence in virtual environments will become more important, too. Dolan-Gavitt et al. (2011b) have pointed out that experiences in memory investigations will be vital for these tasks. However, while researchers succeeded in automating VMI operations to a significant degree (Dolan-Gavitt et al., 2011a; Inoue et al., 2011a), especially the problem of *inconsistent static auditing* remains open that occurs when analysts attempt examining a machine in non-quiescent mode (Nance et al., 2009). Likewise, for *anti-forensic* attacks on VMI applications (see Bahram et al., 2010), suitable countermeasures have yet to be found.

### 7.1.4 Development of Adequate Data Aggregation, Presentation, and Visualization Concepts

Last but not least, as we have argued in an earlier work, "while research in the area of memory forensics has mainly concentrated on identifying, finding, and extracting important data fragments to date, properly correlating and documenting the respective information has been widely neglected" (Vömel and Lenz, 2013, p. 123). Garfinkel (2010, p. S68) also criticizes that "[t]oday's tools were designed to help examiners find specific pieces of evidence, not to assist in investigations" and that these tools "can (sometimes) work with (...) several terabytes of data, but (...) cannot assemble terabytes of data into a concise report". For these reasons, it is necessary to improve current presentation and visualization concepts in order to better support analysts with the completion of a case. With the *rkfinder* plug-in illustrated in Chapter 6, we have already exemplified how information from different sources may be aggregated to create a more concise picture of a computer's state. Teelink and Erbacher (2006), on the other hand, have outlined a method for processing mass data and discovering suspicious elements and outliers with the help of abstract *tree maps* and different color schemes. Finally, Beebe (2009, p. 29) has suggested more focusing on *temporal* instead of hierarchical relationships when examining forensic artifacts. We believe that such approaches may be adopted for memory analysis-related tasks as well and will lead to the development of even more powerful solutions in the future.

# Bibliography


AccessData (2012). FTK Imager. https://ad-pdf.s3.amazonaws.com/Imager_3.1.2_RN.pdf, 2012.

ACPI Promoters Corporation (2011). Advanced Configuration and Power Interface Specification – Revision 5.0. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf, 2011.

Aggarwal, Gaurav; Bursztein, Elie; Jackson, Collin; Boneh, Dan (2010). An Analysis of Private Browsing Modes in Modern Browsers. In *Proceedings of the USENIX Security Symposium*, 2010.

Altheide, Cory; Carvey, Harlan (2011). *Digital Forensics with Open Source Tools*. Syngress Publishing, 2011.

Alvarez, Victor M.; Wiacek, Mike (2013). Yara – A Malware Identification and Classification Tool. http://code.google.com/p/yara-project/, 2013.

Anderson, Ross (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2001.

Apple Inc. (2013). About FileVault 2. http://support.apple.com/kb/HT4790, 2013.

Aquilina, James M.; Casey, Eoghan; Malin, Cameron H. (2008). *Malware Forensics: Investigating and Analyzing Malicious Code*. Syngress Publishing, 2008.

ArxSys (2009). DFF: Digital Forensics Framework. http://www.digital-forensic.org/, 2009.

ArxSys (2012). Introduction to DFF. http://wiki.digital-forensic.org/index.php/Introduction, 2012.

Association of Chief Police Officers (2007). Good Practice Guide for Computer-Based Electronic Evidence. http://www.7safe.com/electronic_evidence/ACPO_guidelines_computer_evidence_v4_web.pdf, 2007.

Aumaitre, Damien (2009). A Little Journey Inside Windows Memory. *Journal in Computer Virology*, Volume 5(2), pp. 105–117, 2009.

Bahram, Sina; Jiang, Xuxian; Wang, Zhi; Grace, Mike; Li, Jinku; Srinivasan, Deepa; Rhee, Junghwan; Xu, Dongyan (2010). DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, 2010.

Ball, Craig (2005). 6 on Forensics – Six Articles on Computer Forensics for Lawyers. http://www.craigball.com/articles.html, 2005.

Barbosa, Edgar (2005). Finding Some Non-Exported Kernel Variables in Windows XP. `http://www.rootkit.com/vault/Opc0de/GetVarXP.pdf`, 2005.

BBN Technologies (2006). FRED: Forensic RAM Extraction Device. `http://www.ir.bbn.com/~vkawadia/`, 2006.

Becher, Michael; Dornseif, Maximillian; Klein, Christian N. (2005). FireWire – All Your Memory Are Belong To Us. In *Proceedings of the Annual CanSecWest Applied Security Conference*, 2005.

Beebe, Nicole (2009). Digital Forensic Research: The Good, the Bad and the Unaddressed. In *Advances in Digital Forensics V*, IFIP Advances in Information and Communication Technology (pp. 17–36). Springer Boston, 2009.

Beebe, Nicole Lang; Clark, Jan Guynes (2007). Digital Forensic Text String Searching: Improving Information Retrieval Effectiveness by Thematically Clustering Search Results. *Digital Investigation*, Volume 4(S1), pp. 49–54, 2007.

Bejtlich, Richard (2006). Forensically Sound Evidence. `http://taosecurity.blogspot.com/2006/08/forensically-sound-evidence.html`, 2006.

Bellard, Fabrice (2012). QEMU. `http://www.qemu.org/`, 2012.

Benenson, Zinaida; Dewald, Andreas; Eßer, Hans-Georg; Freiling, Felix C.; Müller, Tilo; Moch, Christian; Vömel, Stefan; Schinzel, Sebastian; Spreitzenbarth, Michael; Stock, Ben; Stüttgen, Johannes (2011). Exploring the Landscape of Cybercrime. In *Proceedings of the First SysSec Workshop*, 2011.

Betz, Chris (2006). Memparser Analysis Tool. `http://sourceforge.net/projects/memparser/`, 2006.

Bilby, Darren (2006). Low Down and Dirty: Anti-Forensic Rootkits. In *Proceedings of Ruxcon*, 2006.

Böck, Benjamin (2009). Firewire-Based Physical Security Attacks on Windows 7, EFS and BitLocker. `http://www.securityresearch.at/publications/windows7_firewire_physical_attacks.pdf`, 2009.

Boileau, Adam (2006a). FireWire, DMA & Windows. `http://www.storm.net.nz/projects/16`, 2006.

Boileau, Adam (2006b). Hit by a Bus: Physical Access Attacks with Firewire. In *Proceedings of Ruxcon*, 2006.

Boileau, Adam (2008). winlockpwn. `http://www.storm.net.nz/static/files/winlockpwn`, 2008.

Burdach, Mariusz (2005). An Introduction to Windows Memory Forensic. `http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf`, 2005.

Butler, Jamie (2005). FU Rootkit. `http://www.rootkit.com/board_project_fused.php?did=proj12`, 2005.

Carbone, Richard; Bean, C.; Salois, Martin (2011). An In-Depth Analysis of the Cold Boot Attack. `http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA545078`, 2011.

Carr, Chris (2006). GREPEXEC: Grepping Executive Objects from Pool Memory. `http://uninformed.org/?v=4&a=2&t=pdf`, 2006.

Carrier, Brian (2006). Risks of Live Digital Forensic Analysis. *Communications of the ACM*, Volume 49(2), pp. 56–61, 2006.

Carrier, Brian; Spafford, Eugene H. (2003). Getting Physical with the Digital Investigation Process. *International Journal of Digital Evidence*, Volume 2(2), pp. 1–20, 2003.

Carrier, Brian D.; Grand, Joe (2004). A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation*, Volume 1(1), pp. 50–60, 2004.

Carvey, Harlan (2007). *Windows Forensic Analysis*. Syngress Publishing, 2007.

Carvey, Harlan (2011). *Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry*. Syngress Publishing, 2011.

Case, Andrew; Cristina, Andrew; Marziale, Lodovico; Richard, Golden G.; Roussev, Vassil (2008). FACE: Automated Digital Evidence Discovery and Correlation. *Digital Investigation*, Volume 5(1), pp. S65–S75, 2008.

Casey, Eoghan (2007). What Does "Forensically Sound" Really Mean? *Digital Investigation*, Volume 4(2), pp. 49–50, 2007.

Casey, Eoghan (2010). *Handbook of Digital Forensics and Investigation*. Elsevier Academic Press, 2010.

Casey, Eoghan (2011). *Digital Evidence and Computer Crime – Forensic Science, Computers, and the Internet*. Academic Press, 3rd Edition, 2011.

Casey, Eoghan; Fellows, Geoff; Geiger, Matthew; Stellatos, Gerasimos (2011). The Growing Impact of Full Disk Encryption on Digital Forensics. *Digital Investigation*, Volume 8(2), pp. 129–134, 2011.

Chan, Ellick; Venkataraman, Shivaram; David, Francis; Chaugule, Amey (2010). Forenscope: A Framework for Live Forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

Chan, Ellick; Wan, Winston; Chaugule, Amey; Campbell, Roy (2009). A Framework for Volatile Memory Forensics. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

Chan, Ellick M.; Carlyle, Jeffrey C.; David, Francis M.; Farivar, Reza; Campbell, Roy H. (2008). BootJacker: Compromising Computers Using Forced Restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

Check Point Software Technologies Ltd. (2013). Check Point Full Disk Encryption. `http://www.checkpoint.com/products/full-disk-encryption/index.html`, 2013.

Chen, Peter M.; Noble, Brian D. (2001). When Virtual Is Better Than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001.

Chow, Jim; Pfaff, Ben; Garfinkel, Tal; Rosenblum, Mendel (2005). Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.

Christodorescu, Mihai; Jha, Somesh (2004). Testing Malware Detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004.

Cogswell, Bryce; Russinovich, Mark (2006). RootkitRevealer. `http://technet.microsoft.com/de-de/sysinternals/bb897445.aspx`, 2006.

Cohen, Fred (2011). *Digital Forensic Evidence Examination*. ASP Press, 3rd Edition, 2011.

Cohen, Michael (2008). PyFlag – An Advanced Network Forensic Framework. In *Proceedings of the 8th Annual DFRWS Conference*, 2008.

Cohen, Michael (2012a). Finding the Kernel Debugger Block. `http://scudette.blogspot.de/2012/11/finding-kernel-debugger-block.html`, 2012.

Cohen, Michael (2012b). WinPMEM. `http://scudette.blogspot.de/2012/11/the-pmem-memory-acquisition-suite.html`, 2012.

Computer Security Institute (2011). 15th Annual CSI Computer Crime and Security Survey. `http://gocsi.com/survey`, 2011.

Crazylord (2002). Playing with Windows /dev/(k)mem. *Phrack Magazine*, Volume 59(16), pp. 1–14, 2002.

Darawk (2005). CloakDll. `http://www.darawk.com/code/CloakDll.cpp`, 2005.

Davis, Michael; Bodmer, Sean; Lemasters, Aaron (2010). *Hacking Exposed – Malware & Rootkits*. McGraw Hill, 2010.

Detica (2011). The Cost of Cyber Crime. `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/60943/the-cost-of-cyber-crime-full-report.pdf`, 2011.

Dewald, Andreas; Freiling, Felix C.; Schmitt, Sven; Spreitzenbarth, Michael; Vömel, Stefan (2013). Systematische Probleme und Grenzen der forensischen Informatik. In *Jenseits von 1984 - Datenschutz und Überwachung in der fortgeschrittenen Informationsgesellschaft*. Transcript Verlag, 2013.

Dewald, Andreas; Freiling, Felix C.; Schreck, Thomas; Spreitzenbarth, Michael; Stüttgen, Johannes; Vömel, Stefan; Willems, Carsten (2012). Analyse und Vergleich von BckR2D2-I und II. In *Proceedings of Sicherheit*, 2012.

DFRWS (2005). DFRWS 2005 Forensics Challenge. `http://www.dfrws.org/2005/challenge/index.shtml`, 2005.

Dolan-Gavitt, Brendan (2007a). The VAD Tree: A Process-Eye View of Physical Memory. *Digital Investigation*, Volume 4(1), pp. 62–64, 2007.

Dolan-Gavitt, Brendan (2007b). VADTools. `http://vadtools.sourceforge.net/`, 2007.

Dolan-Gavitt, Brendan (2008a). Cell Index Translation. `http://moyix.blogspot.com/2008/02/cell-index-translation.html`, 2008.

Dolan-Gavitt, Brendan (2008b). Finding Kernel Global Variables in Windows. `http://moyix.blogspot.com/2008/04/finding-kernel-global-variables-in.html`, 2008.

Dolan-Gavitt, Brendan (2008c). Forensic Analysis of the Windows Registry in Memory. *Digital Investigation*, Volume 5(1), pp. S26–S32, 2008.

Dolan-Gavitt, Brendan (2008d). Linking Processes to Users. `http://moyix.blogspot.com/2008/08/linking-processes-to-users.html`, 2008.

Dolan-Gavitt, Brendan; Leek, Tim; Zhivich, Michael; Giffin, Jonathon T.; Lee, Wenke (2011a). Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

Dolan-Gavitt, Brendan; Payne, Bryan; Lee, Wenke (2011b). *Leveraging Forensic Tools for Virtual Machine Introspection*. Technical Report, Georgia Institute of Technology, 2011.

Dolan-Gavitt, Brendan; Srivastava, Abhinav; Traynor, Patrick; Giffin, Jonathon (2009). Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

Dornseif, Maximillian; Becher, Michael (2004). Feuriges Hacken – Spaß mit Firewire. In *Proceedings of the 21st Chaos Communication Congress*, 2004.

E-Soft Inc. (2009). Web Server Survey. `https://secure1.securityspace.com/s_survey/data/200907/index.html`, 2009.

Falliere, Nicolas; Murchu, Liam O.; Chien, Eric (2011). W32.Stuxnet Dossier. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`, 2011.

Farmer, Dan; Venema, Wietse (2005a). *Forensic Discovery*. Addison Wesley, 2005.

Farmer, Dan; Venema, Wietse (2005b). memdump – Memory Dumper for UNIX-Like Systems. `http://www.porcupine.org/forensics/tct.html`, 2005.

Foundstone, Inc. (2000). FPort. `http://www.foundstone.com/us/resources/proddesc/fport.htm`, 2000.

Freiling, Felix C.; Schwittay, Bastian (2007). Towards Reliable Rootkit Detection in Live Response. In *Proceedings of IT Incident Management & IT Forensics (IMF)*, 2007.

Gao, Yuhang; Cao, Tianjie (2010). Memory Forensics for QQ from a Live System. *Journal of Computers*, Volume 5(4), pp. 541–548, 2010.

Garcia, Gabriela Limon (2007). *Forensic Physical Memory Analysis: An Overview of Tools and Techniques*. Technical Report, Helsinki University of Technology, 2007.

Garfinkel, Simson L. (2010). Digital Forensics Research: The Next 10 Years. In *Proceedings of the 10th Annual DFRWS Conference*, 2010.

Garfinkel, Tal; Rosenblum, Mendel (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2003.

Garner, George M. (2007). KnTTools with KnTList. `http://gmgsystemsinc.com/knttools/`, 2007.

Giacobbi, Giovanni (2006). The GNU Netcat Project. `http://netcat.sourceforge.net/`, 2006.

Gladyshev, Pavel; Almansoori, Afrah (2010). Reliable Acquisition of RAM Dumps from Intel-Based Apple Mac Computers over FireWire. In *Proceedings of the 2nd International ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, 2010.

GMER (2013). GMER – Rootkit Detector and Remover. `http://www.gmer.net/`, 2013.

GMG Systems, Inc. (2007). KnTTools with KnTList. `http://gmgsystemsinc.com/knttools/`, 2007.

Gurkok, Cem (2012). The Analysis of Process Token Privileges. `http://volatility-labs.blogspot.de/2012/10/omfw-2012-analysis-of-process-token.html`, 2012.

Gutmann, Peter (2001). Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

Halderman, J. Alex; Schoen, Seth D.; Heninger, Nadia; Clarkson, William; Paul, William; Calandrino, Joseph A.; Feldman, Ariel J.; Appelbaum, Jacob; Felten, Edward W. (2008). Lest We Remember: Cold-Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

Hameed, CC (2007). What Is the Page File for Anyway? `http://blogs.technet.com/b/askperf/archive/2007/12/14/what-is-the-page-file-for-anyway.aspx`, 2007.

Hargreaves, Christopher; Chivers, Howard (2008). Recovery of Encryption Keys from Memory Using a Linear Scan. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, 2008.

Haruyama, Takahiro; Suzuki, Hiroshi (2012). One-Byte Modification for Breaking Memory Forensic Analysis. `http://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf`, 2012.

Hauenstein, Thomas; Vömel, Stefan (2013). Possibilities for Extracting Traces of Social Networking Sites from Volatile Memory, 2013.

HBGary (2013). FastDump – A Memory Acquisition Tool. `http://www.hbgary.com/fastdump-pro`, 2013.

Hejazi, S.M.; Talhia, C.; Debbabi, M. (2009). Extraction of Forensically Sensitive Information from Windows Physical Memory. *Digital Investigation*, Volume 6(1), pp. S121–S131, 2009.

Heninger, Nadia; Shacham, Hovav (2009). Reconstructing RSA Private Keys from Random Key Bits. *Cryptology ePrint Archive*, Volume 2008(510), pp. 1–17, 2009.

Hoglund, Greg (2008). The Value of Physical Memory for Incident Response. `http://www.hbgary.com/wp-content/themes/blackhat/images/the-value-of-physical-memory-for-incident-response.pdf`, 2008.

Hoglund, Greg; Butler, James (2005). *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005.

HolyFather (2005). Hacker Defender. `http://www.hxdef.org/download/hxdef100r.zip`, 2005.

Honeynet Project (2004). *Know your Enemy – Learning about Security Threats*. Addison Wesley, 2004.

Hoog, Andrew (2011). *Android Forensics: Investigation, Analysis, and Mobile Security for Google Android*. Syngress Publishing, 2011.

Hulton, David (2006). Cardbus Bus-Mastering: 0wning the Laptop. In *Proceedings of ShmooCon*, 2006.

I.M.Weasel (2006). How to REALLY REALLY Hide from the SC Manager. `http://www.rootkit.com/newsread.php?newsid=419`, 2006.

Inoue, Hajime; Adelstein, Frank; Donovan, Matthew; Brueckner, Steve (2011a). Automatically Bridging the Semantic Gap Using C Interpreter. In *Proceedings of the Annual Symposium on Information Assurance*, 2011.

Inoue, Hajime; Adelstein, Frank; Joyce, Robert A. (2011b). Visualization in Testing a Volatile Memory Forensic Tool. *Digital Investigation*, Volume 8(1), pp. S42–S51, 2011.

Intel Corporation (2013). Intel ® 64 and IA-32 Architectures Software Developer's Manual. `http://download.intel.com/products/processor/manual/325462.pdf`, 2013.

Ionescu, Alex (2005). Getting Kernel Variables from KdVersionBlock, Part 2. `http://www.rootkit.com/newsread.php?newsid=153`, 2005.

Jacob, Solal (2010). Digital Forensics Research Workshop Challenge 2010 – Analysis Report. `http://sandbox.dfrws.org/2010/jacob/dfrws-2010-challenge-analysis.pdf`, 2010.

Kaplan, Brian (2007). RAM Is Key – Extracting Disk Encryption Keys from Volatile Memory. Master's Thesis, Carnegie Mellon University, 2007.

Karayianni, Stavroula; Katos, Vasilios (2011). Practical Password Harvesting from Volatile Memory. In *Proceedings of the 7th International Conference on Global Security Safety and Sustainability*, 2011.

Kdm (2004). NTIllusion: A Portable Win32 Userland Rootkit. *Phrack Magazine*, Volume 11(62), pp. 1–26, 2004.

King, Samuel T.; Chen, Peter M.; Wang, Yi-Min; Verbowski, Chad; Wang, Helen J.; Lorch, Jacob R. (2006). SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

Klein, Tobias (2006a). All Your Private Keys Are Belong to Us – Extracting RSA Private Keys and Certificates from Process Memory. `http://trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf`, 2006.

Klein, Tobias (2006b). Process Dumper. `http://www.trapkit.de/research/forensic/pd/index.html`, 2006.

Kornblum, Jesse D. (2006). Exploiting the Rootkit Paradox with Windows Memory Analysis. *International Journal of Digital Evidence*, Volume 5(1), pp. 1–5, 2006.

Kornblum, Jesse D. (2007). Using Every Part of the Buffalo in Windows Memory Analysis. *Digital Investigation*, Volume 4(1), pp. 24–29, 2007.

Kornblum, Jesse D. (2009). When I'm Sixty Four (Bits). `http://jessekornblum.com/publications/mtn0901.pdf`, 2009.

Laboratory of Cryptography and System Security (2012). *sKyWIper (a.k.a. Flame a.k.a. Flamer): A Complex Malware for Targeted Attacks*. Technical Report, Budapest University of Technology and Economics, 2012.

Lamport, Leslie (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Volume 21(7), pp. 558–565, 1978.

Lempereur, Brett; Merabti, Madjid; Shi, Qi (2010). Pypette: A Framework for the Automated Evaluation of Live Digital Forensic Techniques. `http://www.cms.livjm.ac.uk/pgnet2010/MakeCD/Papers/2010073.pdf`, 2010.

Libster, Eugene; Kornblum, Jesse D. (2008). A Proposal for an Integrated Memory Acquisition Mechanism. *ACM SIGOPS Operating Systems Review*, Volume 42(3), pp. 14–20, 2008.

Ligh, Michael H. (2011a). Investigating Windows Threads with Volatility. `http://mnin.blogspot.de/2011/04/investigating-windows-threads-with.html#!/2011/04/investigating-windows-threads-with.html`, 2011.

Ligh, Michael H. (2011b). Stuxnet's Footprint in Memory with Volatility 2.0. `http://mnin.blogspot.de/2011/06/examining-stuxnets-footprint-in-memory.html`, 2011.

Ligh, Michael H. (2012). Flame & Volatility. `http://mnin.blogspot.de/2012/06/quickpost-flame-volatility.html`, 2012.

Ligh, Michael H.; Adair, Steven; Hartstein, Blake; Richard, Matthew (2010). *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing, 2010.

Linn, Cullen; Debray, Saumya (2003). Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

Maartmann-Moe, Carsten (2012). Adventures with Daisy in Thunderbolt-DMA-Land: Hacking Macs through the Thunderbolt Interface. `http://www.breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface/`, 2012.

Maartmann-Moe, Carsten (2013). Inception. `http://www.breaknenter.org/projects/inception/`, 2013.

Maartmann-Moe, Carsten; Thorkildsen, Steffen E.; Årnes, Andr (2009). The Persistence of Memory: Forensic Identification and Extraction of Cryptographic Keys. *Digital Investigation*, Volume 6(S1), pp. S132–S140, 2009.

Maclean, Nicholas Paul (2006). Acquisition and Analysis of Windows Memory. Master's Thesis, University of Strathclyde, Glasgow, 2006.

Mandiant (2011). Memoryze. `http://www.mandiant.com/products/free_software/memoryze/`, 2011.

ManTech CSI, Inc. (2009). mdd. `http://sourceforge.net/projects/mdd/`, 2009.

Martignoni, Lorenzo; Fattori, Aristide; Paleari, Roberto; Cavallaro, Lorenzo (2010). Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2010.

Mattern, Friedemann (1989). Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.

Microsoft Corporation (2011). Windows Feature Lets You Generate a Memory Dump File by Using the Keyboard. `http://support.microsoft.com/?scid=kb%3Ben-us%3B244139&x=5&y=9`, 2011.

Microsoft Corporation (2012a). Overview of Memory Dump File Options for Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2. `http://support.microsoft.com/?scid=kb%3Ben-us%3B254649&x=13&y=5`, 2012.

Microsoft Corporation (2012b). Windows Registry Information for Advanced Users. `http://support.microsoft.com/kb/256986/`, 2012.

Microsoft Corporation (2013a). BitLocker Overview. `http://technet.microsoft.com/en-us/library/hh831713.aspx`, 2013.

Microsoft Corporation (2013b). Device\PhysicalMemory Object. `http://technet.microsoft.com/en-us/library/cc787565%28v=ws.10%29.aspx`, 2013.

Microsoft Corporation (2013c). ExAllocatePoolWithTag Routine. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff544520%28v=vs.85%29.aspx`, 2013.

Microsoft Corporation (2013d). Forcing a System Crash from the Keyboard. `http://support.microsoft.com/?scid=kb%3Ben-us%3B254649&x=13&y=5`, 2013.

Microsoft Corporation (2013e). MmMapIoSpace Routine (Windows Drivers). `http://msdn.microsoft.com/en-us/library/windows/hardware/ff554618%28v=vs.85%29.aspx`, 2013.

Milković, Luka (2012). Defeating Windows Memory Forensics. `http://events.ccc.de/congress/2012/Fahrplan/events/5301.en.html`, 2012.

Miller, Matt (2004). Metasploit's Meterpreter. `http://dev.metasploit.com/documents/meterpreter.pdf`, 2004.

Mocas, Sarah (2004). Building Theoretical Underpinnings for Digital Forensics Research. *Digital Investigation*, Volume 1(1), pp. 61–68, 2004.

Moore, David; Paxson, Vern; Savage, Stefan; Shannon, Colleen; Staniford, Stuart; Weaver, Nicholas (2003). Inside the Slammer Worm. *IEEE Security and Privacy*, Volume 1(3), pp. 33–39, 2003.

Movall, Paul; Nelson, Ward; Wetzstein, Shaun (2005). Linux Physical Memory Analysis. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

Müller, Tilo; Dewald, Andreas; Freiling, Felix (2011). TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

Müller, Tilo; Dewald, Andreas; Freiling, Felix C. (2010). AESSE: A Cold-Boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, 2010.

Müller, Tilo; Taubmann, Benjamin; Freiling, Felix (2012). TreVisor – OS-Independent Software-Based Full Disk Encryption Secure Against Main Memory Attacks. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2012.

Murr, Mike (2006). "Forensically Sound Duplicate". `http://www.forensicblog.org/2006/08/02/forensically-sound-duplicate/`, 2006.

Myers, Michael; Youndt, Stephen (2007). An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits. `http://download.harris.com/app/public_download.asp?fid=2237`, 2007.

Nance, Kara; Bishop, Matt; Hay, Brian (2009). Investigating the Implications of Virtual Machine Introspection for Digital Forensics. In *Proceedings of the International Conference on Availability, Reliability and Security*, 2009.

Net Applications (2013). Desktop Top Operating System Share Trend. `https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=0`, 2013.

Noone, Scott (2009). Open Handles List. `http://analyze-v.com/?p=95`, 2009.

Ntldr (2013). DiskCryptor – Open Source Partition Encryption Solution. `http://diskcryptor.net/wiki/Main_Page/en`, 2013.

Okolica, James; Peterson, Gilbert (2010a). A Compiled Memory Analysis Tool. In *Advances in Digital Forensics VI*, IFIP Advances in Information and Communication Technology (pp. 195–204). Springer Boston, 2010.

Okolica, James; Peterson, Gilbert L. (2010b). Windows Operating Systems Agnostic Memory Analysis. *Digital Investigation*, Volume 7(1), pp. S48–S56, 2010.

Okolica, James; Peterson, Gilbert L. (2011). Windows Driver Memory Analysis: A Reverse Engineering Methodology. *Computer & Security*, Volume 30(8), pp. 770–779, 2011.

Olajide, Funminiyi; Savage, Nick (2011). Extraction of User Information by Pattern Matching Techniques in Windows Physical Memory. In *Digital Enterprise and Information Systems*, Communications in Computer and Information Science (pp. 449–465). Springer Boston, 2011.

Pan, Jianfeng (2005). IceSword. `http://www.xfocus.net/tools/200509/1085.html`, 2005.

Panholzer, Peter (2008). Physical Security Attacks on Windows Vista. `http://dl.packetstormsecurity.net/papers/win/Vista_Physical_Attacks.pdf`, 2008.

Patterson, David A. (2004). Latency Lags Bandwith. *Communications of the ACM*, Volume 47(10), pp. 71–75, 2004.

Petroni, Nick L.; Fraser, Timothy; Molina, Jesus; Arbaugh, William A. (2004). Copilot – A Coprocessor-Based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

Piegdon, David R.; Pimenidis, Lexi (2007). Targeting Physically Addressable Memory. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2007.

Ponemon Institute (2012a). 2012 Cost of Cyber Crime Study: United States. `http://www.hpenterprisesecurity.com/news/download/2012-cost-of-cyber-crime-study-united-states`, 2012.

Ponemon Institute (2012b). The TCO for Full Disk Encryption. `http://www.winmagic.com/ponemonstudy`, 2012.

Promoters of the 1394 Open HCI (2010). 1394 Open Host Controller Interface Specification – Release 1.1. `http://support.microsoft.com/kb/256986/`, 2010.

Prosise, Chris; Mandia, Kevin (2003). *Incident Response & Computer Forensics*. McGraw Hill, 2nd Edition, 2003.

Rare Ideas (2013). PortableApps.com – Portable Software for USB, Portable and Cloud Drives. `http://portableapps.com/`, 2013.

Reina, Alessandro; Fattori, Aristide; Pagani, Fabio; Cavallaro, Lorenzo; Bruschi, Danilo (2012). When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

Ring, Sandra; Cole, Eric (2004). Volatile Memory Computer Forensics to Detect Kernel Level Compromise. In *Proceedings of the 6th International Conference on Information and Communications Security (ICICS)*, 2004.

Roussev, Vassil; Richard III, Golden G. (2004). Breaking the Performance Wall: The Case for Distributed Digital Forensics. In *Proceedings of the 4th Annual DFRWS Conference*, 2004.

Ruff, Nicolas (2008). Windows Memory Forensics. *Journal in Computer Virology*, Volume 4(2), pp. 83–100, 2008.

Russinovich, Mark E. (1999). MmGetPhysicalMemoryRanges Function. *The System Internals Newsletter*, Volume 1(5), 1999.

Russinovich, Mark E. (2008). Pushing the Limits of Windows: Physical Memory. `http://blogs.technet.com/b/markrussinovich/archive/2008/07/21/3092070.aspx`, 2008.

Russinovich, Mark E. (2011). TCPView. `http://technet.microsoft.com/en-us/sysinternals/bb897437.aspx`, 2011.

Russinovich, Mark E.; Solomon, David A.; Ionescu, Alex (2009). *Microsoft Windows Internals*. Microsoft Press, 5th Edition, 2009.

Rutkowska, Joanna (2006). Subverting Vista Kernel for Fun and Profit. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`, 2006.

Rutkowska, Joanna (2007). Beyond The CPU: Defeating Hardware Based RAM Acquisition (Part I: AMD Case). `http://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf`, 2007.

Sang, Fernand Lone; Nicomette, Vincent; Deswarte, Yves (2011). I/O Attacks in Intel PC-based Architectures and Countermeasures. In *Proceedings of the First SysSec Workshop*, 2011.

Saout, Christophe (2006). dm-crypt: A Device-Mapper Crypto Target. `http://www.saout.de/misc/dm-crypt/`, 2006.

Savoldi, Antonio; Gubian, Paolo (2008). Towards the Virtual Memory Space Reconstruction for Windows Live Forensic Purposes. In *Proceedings of Systematic Approaches to Digital Forensic Engineering (SADFE)*, 2008.

Savoldi, Antonio; Gubian, Paolo; Echizen, Isao (2010). Uncertainty in Live Forensics. In *Advances in Digital Forensics VI*, IFIP Advances in Information and Communication Technology (pp. 171–184). Springer Boston, 2010.

Schatz, Bradley (2007a). BodySnatcher: Towards Reliable Volatile Memory Acquisition by Software. *Digital Investigation*, Volume 4(S1), pp. 126–134, 2007.

Schatz, Bradley (2007b). Recent Developments in Volatile Memory Forensics. `http://www.schatzforensic.com.au/presentations/BSchatz-CERT-CSD2007.pdf`, 2007.

Schatz, Bradley (2010). Finding Object Roots in Vista (KPCR). `http://blog.schatzforensic.com.au/2010/07/finding-object-roots-in-vista-kpcr/`, 2010.

Schreiber, Sven B. (2001). *Undocumented Windows 2000 Secrets – A Programmer's Cookbook*. Addison Wesley, 2001.

Schuster, Andreas (2006a). DMP File Structure. `http://computer.forensikblog.de/en/2006/03/dmp_file_structure.html`, 2006.

Schuster, Andreas (2006b). `_DISPATCHER_HEADER` Structure. `http://computer.forensikblog.de/en/2006/02/dispatcher-header.html`, 2006.

Schuster, Andreas (2006c). Pool Allocations as an Information Source in Windows Memory Forensics. In *Proceedings of IT-Incident Management & IT-Forensics (IMF)*, 2006.

Schuster, Andreas (2006d). Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Investigation*, Volume 3(1), pp. 10–16, 2006.

Schuster, Andreas (2008a). 64bit Crash Dumps. `http://computer.forensikblog.de/en/2008/02/64bit_crash_dumps.html`, 2008.

Schuster, Andreas (2008b). The Impact of Microsoft Windows Pool Allocation Strategies on Memory Forensics. *Digital Investigation*, Volume 5(S1), pp. S58–S64, 2008.

Schuster, Andreas (2008c). PTFinder. `http://computer.forensikblog.de/files/ptfinder/ptfinder-collection-current.zip`, 2008.

Schuster, Andreas (2009a). Linking File Objects to Processes. `http://computer.forensikblog.de/en/2009/04/linking-file-objects-to-processes.html`, 2009.

Schuster, Andreas (2009b). Scanning for File Objects. `http://computer.forensikblog.de/en/2009/04/scanning-for-file-objects.html`, 2009.

SECUDE AG (2012). US Full Disk Encryption 2011 Survey. `http://www.secude.com/us/download/?fid=14`, 2012.

SecurStar Corporation (2008). Disk Encryption Software. `http://www.securstar.com/disk_encryption.php`, 2008.

Shamir, Adi; van Someren, Nicko (1999). Playing Hide and Seek with Stored Keys. In *Proceedings of the Third International Conference on Financial Cryptography*, 1999.

Shipley, Todd G.; Reeve, Henry R. (2006). Collecting Evidence from a Running Computer – A Technical and Legal Primer for the Justice Community. `http://www.hbgary.com/wp-content/themes/blackhat/images/collectevidenceruncomputer.pdf`, 2006.

Sikorski, Michael; Honig, Andrew (2012). *Practical Malware Analysis – The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

Silberman, Peter; C.H.A.O.S. (2006). FUTo. `http://uninformed.org/?v=3&a=7&t=sumry`, 2006.

Simon, Matthew; Slay, Jill (2009). Enhancement of Forensic Computing Investigations through Memory Forensic Techniques. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2009.

Simon, Matthew; Slay, Jill (2010). Recovery of Skype Application Activity Data from Physical Memory. In *Proceedings of the Fifth International Conference on Availability, Reliability and Security (ARES)*, 2010.

Simon, Matthew; Slay, Jill (2011). Recovery of Pidgin Chat Communication Artefacts from Physical Memory. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*, 2011.

Smith, James E.; Nair, Ravi (2005). The Architecture of Virtual Machines. *Journal of Computer*, Volume 38(5), pp. 32–38, 2005.

Solomon, Jason; Huebner, Ewa; Bem, Derek; Szezynska, Magdalena (2007). User Data Persistence in Physical Memory. *Digital Investigation*, Volume 4(2), pp. 68–72, 2007.

Sophos Ltd. (2013). SafeGuard Easy. `http://www.sophos.com/en-us/products/encryption/safeguard-easy.aspx`, 2013.

Sparks, Sherri; Butler, Jamie (2005). Shadow Walker – Raising the Bar for Rootkit Detection. `http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf`, 2005.

Spitzner, Lance (2003a). Definitions and Value of Honeypots. `http://www.spitzner.net/honeypots.html`, 2003.

Spitzner, Lance (2003b). Moving Forward with Defintion of Honeypots. `http://www.securityfocus.com/archive/119/321957/30/0/threaded`, 2003.

Stephenson, Peter (2000). *Investigating Computer-Related Crime: Handbook for Corporate Investigators*. CRC Press, 2000.

Stevens, Richard M.; Casey, Eoghan (2010). Extracting Windows Command Line Details from Physical Memory. *Digital Investigation*, Volume 7(1), pp. S57–S63, 2010.

Stover, Sam; Dickerson, Matt (2005). Using Memory Dumps in Digital Forensics. *;login: The USENIX Magazine*, Volume 30(6), pp. 43–48, 2005.

Stüttgen, Johannes; Cohen, Michael (2013). Anti-Forensic Resilient Memory Acquisition. In *Proceedings of the 13th Annual DFRWS Conference*, 2013.

Su, Zhen; Wang, LianHai (2011). Evaluating the Effect of Loading Forensic Tools on the Volatile Memory for Digital Evidences. In *Seventh International Conference on Computational Intelligence and Security*, 2011.

Suiche, Matthieu (2008). Exploiting Windows Hibernation. In *Proceedings of Europol High Tech Crime Expert Meeting*, 2008.

Suiche, Matthieu (2009a). Reply to HBGary. `http://www.msuiche.net/2009/11/16/reply-to-hbgary-and-personal-notes/`, 2009.

Suiche, Matthieu (2009b). Win32dd. `http://www.msuiche.net/tools/win32dd-v1.2.1.20090106.zip`, 2009.

Suiche, Matthieu (2010). Advanced Mac OS Physical Memory Analysis. `http://www.blackhat.com/presentations/bh-dc-10/Suiche_Matthieu/Blackhat-DC-2010-Advanced-Mac-OS-X-Physical-Memory-Analysis-wp.pdf`, 2010.

Suiche, Matthieu (2013). MoonSols Windows Memory Toolkit. `http://moonsols.com/product`, 2013.

Sutherland, Iain; Evans, Jon; Tryfonas, Theodore; Blyth, Andrew (2008). Acquiring Volatile Operating System Data Tools and Techniques. *ACM SIGOPS Operating Systems Review*, Volume 42(3), pp. 65–73, 2008.

Sylve, Joe; Case, Andrew; Marziale, Lodovico; Richard, Golden G. (2012). Acquisition and Analysis of Volatile Memory from Android Devices. *Digital Investigation*, Volume 8(3), pp. 175–184, 2012.

Symantec Corporation (2013). Symantec Drive Encryption. `http://www.symantec.com/drive-encryption`, 2013.

Teelink, Sheldon; Erbacher, Robert F. (2006). Improving the Computer Forensic Analysis Process through Visualization. *Communications of the ACM*, Volume 49(2), pp. 71–75, 2006.

The Bochs Project (2013a). Bochs – The Cross Platform IA-32 Emulator. `http://bochs.sourceforge.net/`, 2013.

The Bochs Project (2013b). Save and Restore Simulation. `http://bochs.sourceforge.net/doc/docbook/user/using-save-restore.html`, 2013.

The Bochs Project (2013c). Weird Macros and Other Mysteries. `http://bochs.sourceforge.net/doc/docbook/development/emulator-objects.html`, 2013.

Todd, Adam D.; Benson, J.; Peterson, G. L.; Franz, T.; Stevens, M.; Raines, Richard A. (2007). Analysis of Tools for Detecting Rootkits and Hidden Processes. In *Proceedings of the International Conference for Information Processing*, 2007.

TrueCrypt Foundation (2013). TrueCrypt – Free Open-Source Disk On-The-Fly Encryption. `http://www.truecrypt.org/`, 2013.

Tsow, Alex (2009). An Improved Recovery Algorithm for Decayed AES Key Schedule Images. *Lecture Notes in Computer Science*, 5867, pp. 215–230, 2009.

U.S. Department of Justice (2008). Electronic Crime Scene Investigation: A Guide for First Responders. `http://www.ncjrs.gov/pdffiles1/nij/219941.pdf`, 2008.

U.S. Secret Service (2007). Best Practices For Seizing Electronic Evidence. `https://www.ncjrs.gov/App/publications/Abstract.aspx?id=239359`, 2007.

van Baar, Ruud; Alink, Wouter; van Ballegooij, Alex (2008). Forensic Memory Analysis: Files Mapped in Memory. In *Proceedings of the 8th Annual DFRWS Conference*, 2008.

van de Ven, Arjan (2008). x86: Introduce /dev/mem Restrictions with a Config Option. `http://lwn.net/Articles/267427/`, 2008.

Vidas, Timothy (2006). The Acquisition and Analysis of Random Access Memory. *Journal of Digital Forensic Practice*, Volume 1(4), pp. 315 – 323, 2006.

Vidas, Timothy (2010). Volatile Memory Acquisition via Warm Boot Memory Survivability. In *Proceedings of the 43rd Hawaii International Conference on System Sciences*, 2010.

Vidas, Timothy; Zhang, Chengye; Christin, Nicolas (2011). Toward a General Collection Methodology for Android Devices. In *Proceedings of the 11th Annual DFRWS Conference*, 2011.

Vidstrom, Arne (2002). PMDump. `http://ntsecurity.nu/toolbox/pmdump/`, 2002.

Vidstrom, Arne (2006). Memory Dumping over FireWire – UMA Issues. `http://ntsecurity.nu/onmymind/2006/2006-09-02.html`, 2006.

VMware, Inc. (2013). What Files Make Up a Virtual Machine? `http://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html`, 2013.

Volatile Systems, LLC (2008). The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. `https://www.volatilesystems.com/default/volatility`, 2008.

Volatile Systems, LLC (2012). Volatility User Mailing List. `http://lists.volatilesystems.com/pipermail/vol-users/2012-July/thread.html#470`, 2012.

Volatile Systems, LLC (2013a). Volatility – An Advanced Memory Forensics Framework. `http://code.google.com/p/volatility/`, 2013.

Volatile Systems, LLC (2013b). Volatility Documentation Project. `http://code.google.com/p/volatility/wiki/VolatilityDocumentationProject`, 2013.

Vömel, Stefan; Freiling, Felix C. (2011). A Survey of Main Memory Acquisition and Analysis Techniques for the Windows Operating System. *Digital Investigation*, Volume 8(1), pp. 3–22, 2011.

Vömel, Stefan; Freiling, Felix C. (2012). Correctness, Atomicity, and Integrity: Defining Criteria for Forensically-Sound Memory Acquisition. *Digital Investigation*, Volume 9(2), pp. 125–137, 2012.

Vömel, Stefan; Holz, Thorsten; Freiling, Felix C. (2010). *I'd Like to Pay with Your Visa Card: An Illustration of Illicit Online Trading Activity in the Underground Economy.* Technical Report, University of Mannheim, 2010.

Vömel, Stefan; Lenz, Hermann (2013). Visualizing Indicators of Rootkit Infections in Memory Forensics. In *Proceedings of the 7th International Conference on IT Security Incident Management & IT Forensics (IMF)*, 2013.

Vömel, Stefan; Stüttgen, Johannes (2013). An Evaluation Platform for Forensic Memory Acquisition Software. In *Proceedings of the 13th Annual DFRWS Conference*, 2013.

Waits, Cal; Akinyele, Joseph Ayo; Nolan, Richard; Rogers, Larry (2008). Computer Forensics: Results of Live Response Inquiry vs. Memory Image Analysis. `http://www.sei.cmu.edu`, 2008.

Walters, Aaron; Petroni, Nick L. (2007). Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. In *Proceedings of Black Hat DC*, 2007.

Wang, Jiang; Zhang, Fengwei; Sun, Kun; Stavrou, Angelos (2011a). Firmware-Assisted Memory Acquisition and Analysis Tools for Digital Forensics. In *Proceedings of the Sixth International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, 2011.

Wang, Lianhai; Xu, Lijuan; Zhang, Shuhui (2011b). Network Connections Information Extraction of 64-Bit Windows 7 Memory Images. *Forensics in Telecommunications, Information, and Multimedia*, 56, pp. 90–98, 2011.

White, Andrew; Schatz, Bradley; Foo, Ernest (2012). Surveying the User Space through User Allocations. In *Proceedings of the 12th Annual DFRWS Conference*, 2012.

XShadow (2005). Vanquish v0.2.1. `http://rootkit.com/project.php?id=9`, 2005.

Young, Adam; Yung, Moti (2004). *Malicious Cryptography – Exposing Cryptovirology.* Wiley Publishing, 2004.

Yu, Miao; Lin, Qian; Li, Bingyu; Qi, Zhengwei; Guan, Haibing (2012). Vis: Virtualization Enhanced Live Forensics Acquisition for Native System. *Digital Investigation*, Volume 9(1), pp. 22–33, 2012.

Zhang, Lei; Wang, Lianhai; Zhang, Ruichao; Zhang, Shuhui; Zhou, Yang (2011). Live Memory Acquisition through FireWire. *Forensics in Telecommunications, Information, and Multimedia*, 56, pp. 159–167, 2011.

Zhang, Ruichao; Wang, Lianhai; Zhang, Shuhui (2009). Windows Memory Analysis Based on KPCR. In *Proceedings of the Fifth International Conference on Information Assurance and Security*, 2009.

Zhang, Shuhui; Wang, Lianhai; Zhang, Ruichao; Guo, Qiuxiang (2010). Exploratory Study on Memory Analysis of Windows 7 Operating System. In *Proceedings of the Third International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 2010.

Zhao, Qian; Cao, Tianjie (2009). Collecting Sensitive Information from Windows Physical Memory. *Journal of Computers*, Volume 4(1), pp. 3–10, 2009.

Zovi, Dino Dai (2006). Hardware Virtualization Rootkits. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf`, 2006.