Advances in memory forensics

Fabio Pagani

ADVANCES IN MEMORY FORENSICS

FABIO PAGANI

Thèse de doctorat de Informatique et Réseaux

DIRIGÉE PAR
PROF. DAVIDE BALZAROTTI

PRÉSENTÉE ET SOUTENUE PUBLIQUEMENT LE 09/09/2019

DEVANT UN JURY COMPOSÉ DE:
ENGIN KIRDA,
JUAN CABALLERO,
BRENDAN DOLAN-GAVITT,
AURÉLIEN FRANCILLON,
CLÉMENTINE MAURICE,
MARIANO GRAZIANO

I almost wish I hadn't gone down that rabbit-hole—and yet—and yet—it's rather curious, you know, this sort of life! I do wonder what can have happened to me!

— LEWIS CARROLL IN "ALICE'S ADVENTURES IN WONDERLAND"

ABSTRACT

The adoption of memory forensics - the branch of digital forensics that focuses on extracting artifacts from the volatile memory of a compromised system - is rapidly spreading in cyber-security investigations. One of the main reason is that many artifacts that are extracted from system memory cannot be found elsewhere. Therefore, by combining these findings with the results of network and disk analysis, a forensics analysts can better reconstruct the big picture that describes the evolution and the consequences of a computer incident. However, the field of memory forensics is less than two decades old and therefore still has many open challenges and unanswered questions. This thesis provides a new perspective and proposes new solutions for three of the major problems and limitations that affects the area of memory forensics.

The first contribution studies the effects non-atomic acquisition methods. The root cause of this problem is simple to understand: while the physical memory is acquired, user and kernel processes are running and therefore the content of the memory is changing. For this reason, the resulting memory dump does not represent a consistent state of the memory in a given point in time, but rather a mix of multiple chunks acquired at a distance of tens of seconds.

The second contribution focuses on how to automatically extract a forensics profile from a memory dump. Today, having a valid profile that describes the layout of kernel data structures and the location of certain symbols is a mandatory requirement to perform memory analysis, thus preventing memory forensics to be applied in those scenarios where such profile is not available.

The third and last contribution of this thesis proposes a new method to better design and evaluate the heuristics, better known as *plugins*, that are used to extract information from a memory dump. Nowadays, these plugins are manually written by kernel experts and forensics practitioners. Unfortunately, this manual approach does not provide any guarantee on the quality or on the uniqueness of these rules. For this reasons, this thesis presents a framework that can be used to discover, assess, and compare forensics rules.

ACKNOWLEDGEMENTS

Completing a PhD sets a milestone in the life of a young researcher but it must not be seen as a one-man's endeavor – actually quite the contrary. It requires support, guidance and enthusiasm from several people without whom, the already steep path leading here becomes almost impracticable.

First and foremost I want to express my deepest gratitude to Davide, for raising very high the bar of what I consider being a good advisor. Apart from your constant presence in the last years – knocking on your door has always been enough to immediately have a meeting – I want to thank you for showing me what doing research looks like and how to approach problems with the right mindset. But also thank you for introducing me to climbing... even though I am writing this page with a sling on my right arm!

The colleagues and friends I met during this journey also played a fundamental role. A big thank you to Emanuele for his passion about open-source software, Dario the git-master, Savino the (wannabe) tikz-master, Mariano, Marius, Annalisa, Sebastian, Alessandro, Davide, Andrea, Onur, Merve, Matteo, Giovanni, Leyla, Aurelien and the rest of the group because working, playing CTFs and having fun with you has been an incredible privilege. A personal shout-out goes to Yanick for his always positive attitude and for reminding me that even the road of talented researchers can be quite rough sometimes.

I also want to thank professors Engin Kirda and Juan Caballero for their insightful reviews and comments about this thesis, and Brendan Dolan-Gavitt, Aurélien Francillion, Clémentine Maurice and Mariano Graziano for being part of the thesis committee. I owe you all a favor!

Many thanks to the Eurecom administration – in particular to Sophie and Audrey – for dealing with the insane amount of bureaucracy that comes along with a PhD and therefore saving me from countless of headaches.

Outside the academic world, I want to acknowledge my parents Angela and Roberto, my brother Marco, Alessandra and the rest of the family. Even thought we lived far apart for the last 4 years, I always felt your constant encouragement and this is what drove me to always give the best. I was also lucky to be surrounded by long

lasting friendships. In particular Giorgio, Giulia, Daniele, Teona, Niccolò and Marcello must be mentioned because they did not let me drown in work, but instead always kept me in touch with the reality.

Last but not least, a huge thank you to Alice. Because when I told you of this adventure, you where as ecstatic as I was, even though we both knew we were going to face a lot of sacrifices. For your unconditional love and for always been by my side during these years, I want to dedicate this thesis to you, the love of my life.

CONTENTS

1 INTRODUCTION

The pervasiveness of software in our daily life is by now undeniable and consolidated. Even without being fully aware of it, we rely on pieces of software – that sometimes are executed thousands of kilometers away – to drive planes, to buy stocks and to protect healthcare data. In only a few decades, technology advancements have also completely revolutionized the way human beings interact and share content between themselves. This had several and important implications on how this information is managed, both in terms of privacy and confidentiality but also in terms of the security of the systems storing this data. For this reason, a considerable workforce coming from academia and industry has spent more than a half of a century on these topics. The net result of this effort is that systems are becoming more secure, and defenses are widely spread in personal and enterprise environments. Unfortunately, in a cat-and-mouse game fashion, attackers sharpened their knives and continued to carry on with their malicious activities and breach computer systems. Last year's Symantec Threat Report [Sym18] gives a worrisome overview of these malicious activities, showing how no device or appliance is spared: cloud machines, IoT devices and mobile phones were targeted using different attack vectors. In most of the cases these attacks resulted in serious financial losses for companies and individuals. In others, *leaks* of catastrophic proportions happened, and private information became part of the public domain or was sold by criminal organizations.

A very important lesson in computer security is that learning from past mistakes is crucial to properly harden systems against future threats. To this end, establishing a positive feedback loop between *collected data* and *security defenses* is essential. The first edge – connecting data to defenses – ensures that by analyzing the data related to an incident is possible to identify the root cause of the attack and build stronger defenses. But the other way around is also important: as it is often infeasible to collect a fine-grained trace of a system execution, it is important to

to carefully selected the information to collect. Moreover, to complicate the matter, *time* plays a key role in this equation. In fact, some data is inherently transient – i.e. network traffic – and if not timely collected it is lost forever. Similarly, evidences located in unallocated resources – such as unused disk space – can be overwritten by new data and thus destroyed. The science that deals with the collection, the analysis, and the interpretation of the data related to a computer incident is called *digital forensics*. Historically speaking, digital forensics sprang as a science to aid traditional forensics investigation – i.e. to analyze seized devices – but nowadays its techniques are also part of incident response procedures. Traditionally, the focus of digital forensics has been on the analysis of hard drives and only later it shifted to other components of computer systems. One in particular – the *volatile memory* – is receiving a lot of interest from the incident response community. The first reason of this interest comes from the fact that attackers became more and more aware of the traces they leave on disks, thus an analysis focused only on data at rest often times gives an incomplete view over the malicious behavior. Moreover, many artifacts are not found elsewhere *but* in the volatile memory. For example, by only looking at the disk content is impossible to tell which processes where running or which kernel modules were loaded in the system. This new branch of digital forensics – called *memory forensics* – deals with the analysis of the volatile memory extracted from a running system.

1.1 A QUICK INTRODUCTION TO MEMORY FORENSICS

Memory forensics is only 15 years old and traditionally it has been used as part of post-mortem investigations. In addition, more recently it has also been used as a proactive tool to periodically check computers and look for signs of a possible compromise. In both cases, the first step of memory forensics involves obtaining a copy of the content of the volatile memory (represented by the Memory Dump box of Figure 1.1). After this step is successfully preformed, the analyst has two different ways to proceed. The first one is to treat the memory dump as an unstructured blob of data

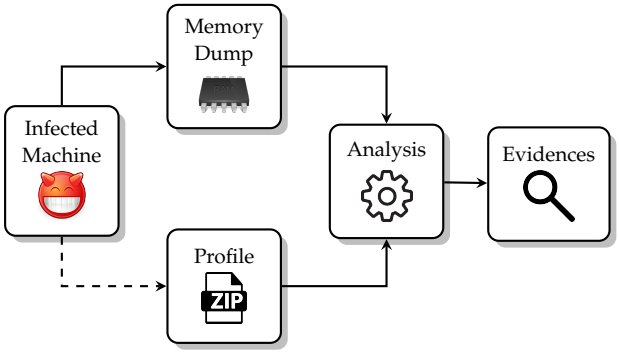


Figure 1.1: Memory Forensics Overview.

and apply carving techniques – often coming from the network and disk analysis domains. In this way, it is possible to extract evidences such as images, open documents, or a chat history. But the true power of memory forensics lies in the second option, namely *structured memory forensics*. In this case, evidences are extracted by parsing and interpreting operating system constructs. This type of analysis is much more powerful than its carving counterpart, because it allows to precisely reconstruct the state of the kernel under analysis and therefore to extract a number of artifacts which would not be accessible otherwise.

1.1.1 *Memory Imaging*

Similarly to the disk domain, memory imaging is the process of obtaining a copy of a system's volatile memory. However, this task is quite challenging given its central role in a computer system. For instance, the standard procedure to image a disk is to halt the machine, extract the disk and connect it to a proper acquisition workstation. On the other hand, this procedure can not be naively ported in the memory acquisition context, because the content of a memory chip rapidly decay when it is disconnected from its socket. For this reason, most of the existing tools acquire the memory while the system is running. However, this bring into play another subtle problem: contrary to disks, imaging the volatile memory

twice never yields the same result. To address this concerns, three criteria has been proposed [VF12] to evaluate the forensics soundness of memory acquisition tools:

- *Correctness*: measures how much a memory dump faithfully represents the RAM content.
- *Atomicity*: measures the impact of the system activity while the dump is taken.
- *Integrity*: measures the footprint of the acquisition process to the memory content.

Memory imaging has been a flourish research area and resulted in a large number of tools that suits different needs and environments, and fulfill – partially or completely – some of the previous criteria. To give an idea of the vast array of choices a digital investigator has, a known website dedicated to digital forensics [Wik] lists more than 20 different tools to do memory acquisition. While this problem can be tackled from different angles, these techniques can be divided in two categories: *hardware* and *software*. Hardware acquisition methods rely on a piece of hardware, i.e. a PCI card, that access the physical memory via Direct Memory Access (DMA) thus bypassing the CPU. Nevertheless, hardware acquisition methods require a kernel driver running in the target system because the Input Output Memory Management Unit (IOMMU) must be properly configured to give the devices full access over the memory [LPF19]. Moreover, these cards must be installed before a security incident happens, because not all of them are hot-pluggable. This limitation is overcome by software methods, which are the most flexible and commonly adopted way to acquire physical memory. Available tools work either from kernel or user space. In the first case, a kernel module is inserted in the running operating system and, since it runs with the same privileges of the kernel itself, it can freely access the entire memory. The only caveat when dumping the memory from kernel space is that the module must differentiate between regions of the physical address space mapped to the RAM and those belonging to devices – because accessing the latter may crash the

system. Fortunately, both Windows and Linux offer a way to access this information. On the other hand, user space methods read the memory from special files exported by the kernel - i.e. `/dev/kmem` in Linux or `\\.\Device\\PhysicalMemory` in Windows. Unfortunately, for security reasons, these files are rarely accessible from user space. Finally, when dealing with virtual machines, an analyst can use hypervisor-specific commands to dump the memory. Sadly, this facility is rarely exported by cloud providers, thus forcing the analyst to fallback to other acquisition strategies [CR17].

1.1.2 *Structured Memory Analysis*

Once the memory dump is acquired and saved on the analyst workstation, the real analysis begins. The core idea behind structured memory forensics is to reconstruct the state of the kernel under analysis, by locating and interpreting kernel structures. The "entry-points" of an analysis are located by using different techniques: some structures are pointed by global variables, other are always at a fixed offset, and some can modeled and carved from raw memory. Once an entry-point structure has been identified, de-referencing the pointers it contains allows the analysis system to discover other structures in a recursive fashion.

While the continuous development of an operating system generally brings new features into its kernel – and new structures that can potentially contain new interesting artifacts – most of the core structures and the way the kernel uses them to save information remains relatively stable across kernel version. For example, in Linux the way processes are organized by the kernel has not changed since version 2.6, released on December 2003. Based on this information, memory forensics frameworks are shipped with several forensics analysis rules – often implemented as *plugins* – to aid the analyst with common investigation tasks. For instance, one of the most common first step in memory analysis consists of running a plugin to list the processes that were running when the memory was acquired. Under Linux, this requires to locate a global variable which contains the root node of the list of processes,

and then to traverse this list. This plugin-based architecture (adopted for instance by the popular open-source frameworks Volatility [Wal07] and Rekall [Coh14]) is very flexible because it effectively relieves the analyst from the burden of understanding every low-level detail when she need to implement a custom analysis plugin. In fact, while the aforementioned procedure to list processes seems straightforward, it involves solving several challenges. For example, the kernel page tables must be located to translate virtual to physical addresses and raw chunks of memory must be interpreted to extract higher-level information. The latter problem, known as bridging the *semantic gap*, afflicts both the forensics and the virtual machine introspection community. In fact, both of them need a very precise and detailed model of the kernel structures under analysis to complete their tasks. In memory forensics terms, this model is called a *profile*. The information needed to create a valid profile vary depending on the operating system, thus the dotted line in Figure 1.1. Building a profile for a Windows kernel is quite straightforward because Microsoft releases the debugging symbols of every kernel on a public symbol server. For example Rekall [Coh14] is able to automatically identify the version of the kernel contained in a memory dump, download the corresponding symbols and create a profile. On the other hand, the Linux kernel can be customized with thousands of configuration options defined by the user at compile time. Since these options affect the layout of kernel structures, a profile needs to be created for the specific version of the Linux kernel running in the system where the memory was acquired.

1.2 CHALLENGES & OPEN PROBLEMS

Memory forensics is a very active and fast paced topic: analysts have to continuously look out for new software and operating system versions to update their tools accordingly. Quite often these advancement gaps can be fulfilled with a small engineering effort, but in other cases they require completely new approaches and techniques.

This thesis focuses on three *fundamental* problems, which are intrinsically tied to memory forensics. What makes these problems even more

interesting to study is the fact that their existence is well known by memory forensics practitioners and researchers, but their implications are often underestimated.

Challenge #1 - Non-Atomic Memory Imaging

The spectrum of choices for memory imaging is very broad but a common trait among existing tools is that the computer is not halted during the acquisition process. This means that while memory pages are read, the operating system and user space programs are running and modifying the memory. As a result, a non-atomic acquisition contains *inconsistencies*, because the dump does not reflect the content of the memory at a precise point in time, but is instead more similar to a long exposure photograph of the memory content. While several research papers have tried to define and quantify this problem, its implications on forensics analysis are still unclear and existing tools do not provide any way to assess the possible consequences of a lack of atomicity.

Challenge #2 - Profile Requirement

The existence of a kernel profile plays a central role in memory forensics, because without it structured memory analysis techniques cannot be applied. A problem with this profile-centered approach is that profiles are tightly associated with a specific kernel build. For example, the very same version of the Linux kernel compiled with two different configurations can result in dramatic changes in the profiles. This problem is effectively limiting the scope of applicability of memory forensics, especially when it comes to analyze IoT devices, network devices or smartphones. In these situations, creating a valid profile can be extremely challenging or even impossible.

Challenge #3 - Forensics Rules

Forensics analysis rules, more commonly known as *plugins* in the Volatility framework, represent the core of structured memory forensics. By traversing kernel structures, these rules reconstruct the state of the kernel

under analysis and extract information which are unlikely to be found elsewhere. For example, Volatility ships with more than 50 plugins to analyze the Windows kernel. A problem of these rules is that they are based on a set of heuristics *manually* defined by experts based on their own judgment and experience.

However, it is unclear how different heuristics that extract the same information can be compared against one another. Is it possible, and under which conditions, for them to reach different conclusions? Is there a set of metrics we can use to assess their quality? And, even more important, is it possible to *automatically* design new and better rules the provide some form of guarantees over their execution?

1.3 CONTRIBUTIONS

This thesis tackles the three aforementioned challenges by making the following three novel contributions.

Introducing the Temporal Dimension to Memory Forensics

The first contribution of this thesis, presented in Chapter 3, focuses on the lack of atomicity in memory dumps. This phenomena, also know as "page smearing", has been known since the dawn of memory forensics [Car15]. The focus of prior research on this problem has been on defining and quantifying the level of atomicity of a memory dump, but never on understanding the consequences on actual forensic analyses. To approach this problem, this thesis introduces a new dimension – called *temporal dimension* – to memory forensics . This dimension gives the analyst a preliminary way to assess the atomicity of the data structures used during an analysis. Moreover, we conducted a set of experiments to show that *page smearing* does not only introduce inconsistencies in page tables but instead it indiscriminately affects any analysis traversing structures located in user and kernel space. The impact of these inconsistencies can be more or less serious, but our experiments show that the outcome of a forensics analysis is often impacted. For example, in some tests an

inconsistency in the kernel structures used to track the memory mappings of a process effectively prevented the analysis to locate the code section of a process.

To limit these negative effects, we propose a new *locality-based* memory acquisition method. All memory acquisition tools acquire the physical pages sequentially which, in other words, means that all pages are treated with the same importance, regardless of their content. By using our locality-based acquisition, memory pages containing tightly-related forensics information – such as the elements of a linked list or the page tables of a process – are acquired instead in a short amount of time. This acquisition schema was sufficient to avoid the inconsistencies we encountered in our experiments.

This contribution has been published in an article appeared in the ACM Transactions on Privacy and Security (TOPS) [PFB19]

Towards an Automated Profile Generation for Memory Forensics

The second contribution of this thesis focus on automatically creating forensics profiles, which are an essential component of memory forensics. A Linux profile contains two separate but equally important pieces of information: the layout of kernel structures and the address of global symbols. While Microsoft Windows profiles can be automatically created, when it comes to Linux manual intervention is needed. The process of creating a profile essentially involves the compilation of a kernel module, out of which the structures layouts are extracted. While this task may appear straightforward, the process has several implicit requirements – including the availability of the kernel configuration options. Moreover, in the last few years, a new threat to memory forensics is arising in the form of kernel structures layout randomization (`RANDSTRUCT`). When this hardening technique is enabled at compile time, the fields of sensitive structures are shuffled. If the randomization seed is deleted, then a profile cannot be created. To overcome these problems, Chapter 4 presents a novel way to automatically reconstruct a profile from a memory dump, without relying on any other non-public information. The core idea behind this approach is that, while the layout information is lost during the compi-

lation process, the generated code retains some traces of the position of each field. More specifically, the displacement used to access structure members reflect the layout of the structure itself. Therefore, structure layouts can be recreated by carefully extracting these displacements from kernel binary code. The presented approach can be conceptually divided in three phases. In the first, the kernel version and the location of kernel global variables are extracted from the memory dump. In the second phase, a custom compiler plugin analyzes the kernel source code and records *where* and *how* kernel structures are used. This information is stored in a set of models, called *access chains* which, in the third and last step, are matched against the kernel binary code extracted from the memory dump. Experimental results show that this system is effectively able to recover the layout of structures used in many fundamental forensics plugins, such as those that list processes, memory mappings, opened files, and network connections information.

The content of this chapter is current under submission at the 41st IEEE Symposium on Security and Privacy (S&P 2020)

Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques

The last contribution of this thesis is presented in Chapter 5 and proposes a new framework to extract and evaluate the heuristics used for forensics analysis. The problem with the current approach is that these rules are manually crafted by experts, following their own personal intuition and knowledge. This is unfortunately not enough to ensure the uniqueness nor the quality of a technique that extracts a given information. The fact that there exists multiple way of reaching the same information is known in the forensics community. For example, the current version of Volatility ships with two different plugins to extract the process list under Linux. However, it is unclear if other (better?) solutions are hidden in the maze of kernel code just waiting to be discovered. A second orthogonal problem is how to define the quality of an heuristic. What does it mean that one is *better* than another? Sadly, today there is no way to compared different techniques.

To answer these questions, Chapter 5 we describe how we built a directed graph of kernel structures where nodes represent kernel structures while edges are pointers from one structure to another. Given this representation, memory forensics plugins can be seen as *paths* – or even better as *subgraphs* – in a graph. Then, to assess the quality of a given heuristic, a set of metrics under which a path can be evaluated need to be defined. In this thesis several metrics related to memory forensics are proposed, but new ones can be easily plugged into our framework. Example of these metrics are those that capture the atomicity of a path, or the one used to express the stability of a path over time. We then study the proprieties of this graph and show three different applications of our framework. In the first, the framework is used to study the quality of current memory forensics plugins under the proposed metrics. In the second, previously unknown techniques for listing running processes are discovered. Finally, in the third and last scenario, we compute the optimal paths for each metric for a particular task.

This contribution has been presented at the 28th Usenix Security Symposium [PB19]

2 RELATED WORK

Structured memory forensics was effectively kickstarted by the DFRWS Forensics Challenge in 2005. Before this challenge, all the results of memory analysis were based on carving techniques and string identification. Over the last decade, the forensics community has grown and focused on many different problems, including the application of live forensics [LV08; Ade06; Jon07], the creation of signatures for kernel data structures [Dol+09; Lin+11; Fen+14], the evaluation of different acquisition methods [VF11; VS13; LPF19], the de-randomization of the kernel memory [GL16], the identification of the kernel version [RAS14; BA18], the extraction of hypervisor-related information [GLB13], the design of a number of different analysis techniques tailored to common user space applications [SS10; Al +11; SS11; Ots+18; BD17; AD07; CR16; Mac13] and the applicability of neural network models to memory forensics [Son+18].

In the next sections we are going to explore some of these papers in more details, with a focus on the studies related to the three challenges we are addressing in this thesis.

2.1 ATOMICITY

Even though the lack of atomicity is mentioned by several studies [HBN09; LK08; MC13; Kor07] only very few works have focused on this topic. The first effort to evaluate the consistency of a memory image was done by Huebner in 2007 [Hue+07]. This work acknowledges that capturing a memory image from a live system can indeed introduce inconsistencies between interconnected objects. To resolve this issue the authors explore the applicability of concepts from the area of orthogonally persistent operating systems to computer forensics. In these systems, the state of the kernel and user space applications is captured and recorded periodically. For this reason, the solution proposed by the authors requires integration in the operating system, thus resulting in no impact to practical systems.

The first systematization of the atomicity concept was published by Vomel and Freiling [VF12] through a formal definition of three criteria, namely *correctness*, *integrity*, and *atomicity*. The authors pointed out that an atomic snapshot is a snapshot which "*does not show any signs of concurrent system activity*". Four years later, Gruhn and Freiling [GF16] returned to the subject, this time evaluating how those three criteria are respected by 12 different acquisition tools. More recently, Case and Richard [CR17] brought back under the spotlight the *page smearing* problem. The authors argue that - with the current acquisition tools - the smearing effect will become more and more common, since nowadays servers are equipped with large amount of RAM that causes longer acquisition time. They also underline how page smearing "*is one of the most pressing issues*" in the field of memory forensics.

The very first software acquisition method that tried to obtain an atomic snapshot is BodySnatcher [Sch07]. The underlining idea of this work is to freeze the running operating system by injecting a small custom acquisition OS. The approach, as acknowledged by the authors, has several severe limitations – including the fact of being very operating system dependent and, in its current implementation, of supporting only systems with a single CPU core. In 2010, Forenscope [Cha+10] took advantage of the data remanence effect in memory chips to reboot the system and divert the boot sequence to the acquisition module. This approach is generally referred as cold boot memory acquisition [Hal+09]. Unfortunately, few years later, an in-depth analysis of cold boot practicability in memory forensics [CBS11] concluded that the data remanence strongly depends both on the chipset and on the memory modules used and that some combinations of components do *not* retain the content of the RAM upon a reboot.

The next attempt towards a sound memory acquisition was done by HyperSleuth [Mar+10], a tool based on the Intel virtualization technology. Working at this high-privileged level, allows HyperSleuth to dump pages using two different strategies: *dump-on-write* and *dump-on-idle*. The first is triggered whenever a page is modified by the guest operating system, while the second whenever the guest itself is in an idle state. While this is resilient against malware which tries to evade the forensics acquisition,

it does not avoid data structure inconsistencies. In 2012 Reina et al. presented `SMMDumper` [Rei+12]. a special firmware running in System Management Mode (SMM). Entering this special mode ensures that the acquisition process is completely atomic, since the operating system has no chance to gain back the control. Even if this tool satisfies all of the three forensics criteria, it suffers from the fact that it cannot be hot-plugged in a running system, and therefore it needs to be pre-installed beforehand. A similar approach has also been proposed by Sun et al. [Sun+15] for the ARM architecture. In this case, the authors propose an acquisition process built on top of the `TrustZone` technology.

While not directly related to our findings, Bhatia et al. [Bha+18] and Saltaformaggio et al. [Sal+16; Sal+15a; Sal+15b] studied how it is possible to create a timeline of past user activities and to extract a number of photographic evidences and past Android app GUIs.

While all the attempts described above are interesting from a research perspective, many require the system to be carefully configured beforehand and none is mature enough to be used in real investigations or production systems. As a consequence, memory acquisition of non-virtualized systems continues to be performed using kernel-level solutions that copy the memory content while the system is running.

2.2 KERNEL GRAPHS

The analysis of kernel objects and their inter-dependencies has attracted the interest of both the security and the forensics community. While the common goal, namely ensuring the integrity of the kernel against malware attacks using the inter-dependencies between kernel object, is shared by the majority of works on this topic, the methods and the tools used to achieve it are often different. Several research papers have also been published on reconstructing and analyzing data structure graphs of user-space applications [Coz+08; Bur+11; Urb+14; Sal+14]. However, most of these techniques are not directly applicable to kernel-level data structures, because they do not take in account the intricacies present in

the kernel, such as resolution of ambiguous pointers to handle custom data structures. For this reason they will not be discussed in this Section.

To better highlight the different approaches, we decided to divide them in two distinct categories. The first one covers approaches which presented the analysis of a *running* kernel. The second category is focusing instead on *static* approaches, which require only a memory snapshot or the OS binary.

Dynamic Analysis

One of the first example of dynamic kernel memory analysis was presented by Rhee et al. in 2010 [Rhe+10]. The tool, named `LiveDM`, places hooks at the beginning and at the end of every memory-related kernel functions, to keep track of every allocation and deallocation event. When these hooks are triggered, the hypervisor notes the address and the size of the allocated kernel object and the call site. The latter information is used, along with the result of an offline static analysis of the kernel source code, to determine the type of the allocated object. A work built on top of `LiveDM` is `SigGraph` [Lin+11]. In this paper the authors generate a signature for each kernel object, based on the pointers contained in the data structure. These signatures are then further refined during a profiling phase, where problematic pointers - such as null pointers - are pruned. The result can then be used by the final user to search for a kernel object in a memory dump. The major concern about signature-based scheme is their uniqueness, that avoids problems related to isomorphism of signatures. The authors found that nearly the 40% of kernel object contains pointers and - among this objects - nearly 88% have unique `SigGraph` signatures. Unfortunately the uniqueness was reported *prior* the dynamic refinement, so it is unclear the percentage of non-isomorphic signatures. For this reasons a kernel graph built using the approach adopted by `SigGraph` would only retrieve a *partial* view of the entire memory graph.

Another work focused on signatures to match kernel objects in memory was done by Dolan-Gavitt et al in 2009 [Dol+09]. The main insight of this work is that while kernel rootkits can modify certain fields of a structure - i.e. to unlink the malicious process from the process list - other fields (called *invariants*) can not be tampered without stopping the mali-

cious behavior or causing a kernel crash. The invariant are determined in a two steps approach. During the first one every access to a data structure is logged, using the stealth breakpoint hypervisor technique [VY05]. Then, the most accessed field identified in the previous step are fuzzed and the kernel behavior is observed. If the kernel crashes then there are high chances that this field can not be modified by a rootkit. On the other hand, if no crash is observed, than the field is susceptible to malicious alteration. The direct results of these two phases is that highly accessed field which result in a crash when fuzzed are good candidates to be used as *strong* signatures. While this approach looks very promising is not easily adaptable to our context since, as also noted by the authors, creating a signature for small structures can be difficult. Furthermore, generating a signature requires to locate at least one *instance* of a structure in memory, which might not be straightforward.

Xuan et al. [XCB09] proposed *Rkprofiler*. This tool combine a trace of read and write operations of malicious kernel code with a pre-processed kernel type graph, to identify the tampered data. The problem of ambiguous pointers is overcome by annotating the type graph with the real target of a list pointer. While it is not clearly stated in the paper, it seems the annotation was manually done. As we discussed in Section 5.4 the Linux kernel uses a large amount of ambiguous pointers, thus making the manual annotation approach not feasible anymore.

While more focused on kernel integrity checks, `OSck` [Hof+11] uses information from the kernel memory allocator (*slab*) to correctly label kernel address with their type. The integrity check are run in a kernel thread, separated by the hypervisor. This two components allow OSck to write custom checkers that are periodically run. While this approach seems promising, only a small subset of frequently-used structures are allocated using *slab* (such as `task_struct` or `vm_area_struct`), and thus is unsuitable for our needs.

Another approach to create a graph of Windows kernel objects is *MACE* [Fen+14]. Using a pointer-constrain model generated from dynamic analysis on the memory allocation functions and unsupervised learning on kernel pointers, MACE is able to correctly label kernel objects found in a memory dump. Once again, while the output of the work is a

kernel object graph for Windows, the application only focuses on rootkit detection.

Static Analysis

One of the most prominent work in this field is KOP [Car+09], and its subsequent refinement MAS [Cui+12]. Very similarly to our approach, the authors use a combination of static and memory analysis techniques respectively on the kernel code and on a memory snapshot. In the first step they build a precise *field-sensitive* points-to graph, which is then used during the memory analysis phase to explore and build the kernel objects graph. Contrary to this solution, ours does not make *any* assumption about the kernel memory allocator. While the Windows kernel has only one allocator, the Linux kernel has three different ones (`slab`, `slub`, and `slob`). Moreover, only a predefined subset of kernel objects are allocated in custom slabs, while the vast majority is sorted in generic slabs based on their size.

Gavitt and Traynor [DT08] also created a graph of kernel structures with the purpose of detecting dummy objects, i.e. false positives objects that are matched by memory scanners. As shown by Williams and Torres [WT14] this attack vector can really complicate memory analysis, because the attacker can inject thousands of fake artifacts to confuse the analyst. To counter this eventuality, Gavitt and Traynor proposed to use the PageRank [BP98] algorithms to discard false positive structures. Their evaluation is promising and shows how this metric is enough to discern between active and inactive processes.

Gu et al. [Gu+14] presented `OS-Sommelier`$^+$, a series of techniques to fingerprint an operating system from a memory snapshot. In particular, one of these techniques is based on the notion of *loop-invariants*: a chain of pointers rooted at a given kernel object that, when dereferenced, points back to the initial object. Once a ground truth is generated from a set of known kernels, this loop invariant *signatures* can be used to fingerprint unknown kernels. Unfortunately the paper does not mention the problem of ambiguous pointers, and we believe our graph generation approach could improve `OS-Sommelier`$^+$ detection.

Finally, as we already discussed, none of the tools to automatically build a graph of kernel objects was publicly available.

2.3 AUTOMATIC PROFILE EXTRACTION

Type inference on binary code has been a very active research topic in the past twenty years. In fact, the process of recovering the type information lost during the compilation process involves several challenges and can be tackled from different angles. The applications that benefit from advances in this field are the most diverse, including *vulnerability detection*, *decompilation*, *binary code reuse*, and *runtime protection mechanisms*. Recently, Caballero and Lin [CL16] have systematized the research in this area, highlighting the different applications, the implementation details, and the results of more than 35 different solutions. Among all, some of these systems are able to recover the layout of records, and in some cases to associate a primitive type (for example `char`, `unsigned long`..) to every element inside a record. Examples of these systems are `Mycroft`, `Rewards`, `DDE`, `TDA`, `Howard`, `SmartDec` and `ObjDigger` [Myc99; LZX10; SSB10; TDC10; SSB11; Fok+11; Jin+14]. Unfortunately, these approaches have limited applicability to our problem because they fundamentally answer a different question. While our system tries to retrieve, for example, the offset of a specific field `X` inside an object `Y`, previous approaches were instead interested in retrieving the types of the fields inside `Y`. This is the difference between locating a given integer field (e.g., a process identifier) among dozen of integer fields within the same data structure.

On the other hand, the forensics community is well aware of this problem and over the years proposed some preliminary solutions. The first attempt at solving this problem was done by Case et al. [CMR10] in 2010 and, quite similarly, by Zhang et al. [ZMW16] in 2016. Their approach is quite straightforward: after locating a set of defined functions, the authors extracted the layout of kernel objects by looking at the disassembly of these functions. While we believe it was a step in the right direction, these approaches had several limitations. First of all, both the functions and the corresponding objects were selected manually. This limited the scalability of the solution, and in fact the authors were only able to manually recover a dozen fields in total - while our

experiments emphasize that volatility uses more than two hundred of them. Moreover, to locate the functions in the memory dump, previous solution rely on the content of `System.map`, therefore suffering from all the problems and limitations we discussed in Section 3.1. Finally, since the authors used a simple pattern-matching to extract the offsets from the disassembled code, those approaches worked only on small functions and only if the instructions emitted by the compiler followed a certain predefined pattern.

Case et al. [CMR10] and Zhang et al. [Zha+17] presented also another approach based on the relationship among global kernel objects. Both authors noted that, for example, the field `comm` of the variable `init_task` always contains the string `swapper` or similarly that the field `mm` of the same variables always points to another global variable (`init_mm`). With this information is trivial to extract offsets, because it is enough to find the starting address of `init_task` and scan the following chunk of memory. Unfortunately, not all the object types in the kernel have a corresponding global variable, thus limiting this approach to only a narrow subset of data structures.

Finally, in 2016 Socała and Cohen [SC16] presented an interesting approach to create a profile on-the-fly, without the need to rely on the compiler toolchain. Their tool, `Layout Expert`, is based on a *Preprocessor Abstract Syntax Tree* of kernel objects, that retains all the information about the `ifdefs`. This special AST is created offline and then specialized to the system under analysis, only when the analyst has access to it. Nevertheless, the specialization process still needs the kernel configuration and the `System.map`, making this technique not applicable to our scenario.

3 INTRODUCING THE TEMPORAL DIMENSION TO MEMORY FORENSICS

3.1 INTRODUCTION

To date most of the research in memory forensics has focused on techniques to overcome the *semantic gap* and reconstruct a faithful picture of the system under analysis. In other words, the main challenge has been to automatically assign individual bytes to the corresponding high-level components, such as running processes, device drivers, and open network connections. This process requires a precise knowledge of the internal data structures used by the target operating system, combined with a set of heuristics to locate and traverse these structures and retrieve the desired information. We call this part the **spatial aspect** of memory analysis, as it deals with the *location* of data objects and with their point-to relationships in the address *space* of the system under analysis.

While this spatial analysis still suffers from a number of open problems [CR17], memory forensics is a mature field and popular tools in this area are routinely used in a large number of investigations.

In this thesis we introduce a second, orthogonal, dimension that we believe plays an equally important role in memory analysis: **time**. If the spatial dimension deals with the precise localization of key data structures, the temporal dimension is concerned with the temporal consistency of the information stored in those structures. This new dimension should not be confused with the one introduced by Saltaformaggio [Sal18] where the authors focused on reconstructing a *timeline* of past user activities. Instead, our dimension is tightly related with the memory acquisition process and on how the content of memory changes during this process. In fact, the most common tools that are regularly used to acquire the physical memory of a running machine are executed while the rest of the system is running, and therefore while the content of the memory itself is rapidly changing. This is the case of software-based solutions, as

well as hardware-based approaches that retrieve memory through DMA. As a result, instead of acquiring a precise snapshot of the memory, these tools obtain some sort of blurred, long-exposure picture of a moving target. This issue is well known among practitioners in the field. Already in 2005, when memory forensics was still in its infancy, Harlan Carvey posted a message to the Security Incidents mailing list [Car15] pointing out that the inability of software collection tools to freeze the memory during the acquisition was going to become a problem for future analysis. Unfortunately, this warning remained largely unexplored by researchers for over a decade – while all resources were dedicated to improve the algorithms used to overcome the semantic gap and recover useful information. As a result, existing tools and techniques do not provide any way to estimate, mitigate, or even simply to understand the presence and the possible impact of errors introduced by the lack of atomicity in the collected dumps.

Only recently, researchers have looked at the lack of atomicity in memory dumps, but previous works have focused only on introducing definitions and confirming that in fact software acquisitions produce non-atomic memory images [GF16; VF12]. In these studies, the authors reported problems of "inconsistent page tables" in 20% of their dumps, likely due to *page smearing* [CR17] and resorted to repeat their experiments when this problem was present. In 2018, Le Berre [Le 18] also confirmed the astonishing result that *about every fifth acquisition results in an unusable memory dump*. Sadly, while this is undeniably one of the *"main obstacles to complete memory acquisition"* [CR17], the forensic community is following a "dump and pray" approach, simply suggesting to collect new snapshots when there is evidence of incorrect results. Unfortunately, this is a luxury that is rarely available in a real investigation.

To mitigate this problem, in this thesis we suggest for the first time that the two dimensions (**spatial** and **temporal**) need to be always *recorded* and *analyzed* together. The analysis of the value of memory objects allows the analyst to reconstruct a picture of the state of the target machine and its applications. The analysis of the time those values were acquired provides instead a way to estimate the confidence that the extracted

information (and therefore the result of the entire analysis) is consistent and correct. Our experiments show that page smearing is just the tip of the iceberg and similar inconsistencies occurs everywhere in the memory acquired by live acquisition techniques. In fact, while it is obvious that memory snapshots taken from *inside* a running system are always non atomic, it is still unclear whether (and how often) this can result in wrong conclusions during an investigation. To answer this question, we present a number of real examples that we use to pinpoint three main types of time-based inconsistencies in a number of real memory analysis tasks. Our experiments show that inconsistencies due to non-atomic acquisitions are very common and affect several data structures. Moreover, these inconsistencies often lead to potential errors in the analysis process.

Knowing the extent of the problem is important, but it does not help to improve the current memory forensic field. Therefore, we argue that memory acquisition tools should provide the user not only with the raw data required for the analysis, but also with the necessary information to clearly estimate the reliability of this data and the possible impact of non-atomic collection on a given forensic task. To follow this recommendation, in Section 3.5 we introduce the temporal dimension to two of the most popular memory forensic tools: the Volatility analysis framework, and the LiME acquisition tool. Our changes allow for the first time to precisely record time information in a memory dump and to transparently support this information during the analysis. Moreover, whenever an atomic collection is not possible, in Section 3.5.3 we discuss a better algorithm to decrease the impact of the acquisition time on the memory analysis process. In particular we propose a simple context-sensitive approach to improve the "local-atomicity" of a number of relevant data structures.

3.2 SPACE AND TIME IN MEMORY ACQUISITION

Tens of different memory acquisition techniques have been proposed to date, all with their combination of strengths and weaknesses. The list includes dedicated hardware solutions [CG04; Cox+18], DMA acquisi-

tion via the FireWire bus [BD04; BDK05], and the use of hibernation files [Ruf08] or crash dumps [Sch06]. Despite this variety of techniques, software-based acquisition solutions (e.g., `LiME` [Syl12], `WinPMem` [Coh12], and `Memoryze` [Man]) remain the tool of choice in any scenario that does not involve emulated or virtual environments – where a consistent snapshot can be acquired from the hypervisor while the underlying OS is frozen.

The first step performed by any kernel-level tool is to retrieve the system memory layout. The layout specifies which memory region are usable by the operating system, and which are reserved for memory-mapped peripherals, such as PCI devices, or for the system BIOS. The only way to retrieve this information is to ask the BIOS itself, which can only be queried during the boot phase, when the system is still in *real mode*. For this reason, OS kernels maintain a copy of this list, which in Linux is pointed by the `iomem_resource` symbol and in Windows is accessible through the `MmGetPhysicalMemoryRanges` API. This step is very important, as any attempt to access reserved regions may result in unpredictable side effects [Lig+14] as a device can sense and react to the read requests generated by the acquisition process and place the system in an unstable state (or force it to crash). Once the memory layout has been retrieved, the acquisition process can finally start. For each region assigned to system RAM, the module *maps* a page of physical memory and stores it in a file. In Linux this is done using the `kmap` kernel API[1].

For the purpose of this paper we are only interested in two aspects of the acquisition process. The first is whether the target system is running while the content of the memory is copied. In the case of OS-based or hardware-based acquisitions, this is indeed the case. Thus, while the memory dump is collected, new processes are spawned, incoming connections are handled, and files are read and written to the disk. The second aspect is the order in which the individual pages are acquired by the tool. To the best of our knowledge, for efficiency reasons all solutions acquire the

1 Since on several architectures the Linux kernel maintains a direct mapping to the entire physical memory, this API just performs an address translation and does not involve the modification of any paging structure [SC13]

memory sequentially, starting from lower to the higher available page.

Once the acquisition process is completed, the real analysis can finally begin. Here the main challenge is the fact that while a memory dump contains a copy of the *physical* pages, the analyst often needs to reason in term of OS-provided abstractions, such as processes and their virtual memory space. To bridge this *semantic gap*, memory forensic tools (such as the popular open source *Volatility* [Wal07] and *Rekall* [Coh14] frameworks) require building a *profile* of the operating system that was running on the target host. Such profiles contain information related to relevant kernel data structures and their positions in memory. While the creation of these profiles and their corresponding analysis, what we call the *spatial analysis*, is an active research area (see for instance Socala and Coen [SC16] and Gavit et al. [Dol+09; Dol+11]), this paper focuses on a different and often overlooked problem: the fact that all existing analysis algorithms are based on the assumption that *all pages were acquired at the same time in a single atomic operation*. Not only this is never the case in the acquisition process we described, but as we will see in Section 3.3 and 3.4, this can have important consequences on the results of the analysis.

Atomic Acquisition and Time-Consistency

An ideal acquisition procedure would collect the entire memory in a single atomic operation – such that for the system under analysis the acquisition appears to be performed instantaneously. For all practical purposes, an acquisition can still be considered atomic if the content of the memory does not change between the beginning and the end of the acquisition process – making the result indistinguishable from an hypothetical snapshot collected in a single operation. Recently, Vomel et Freiling [VF12] proposed a more permissive definition of Atomicity, inspired from the theory of distributed systems. This atomicity, which we call *Causal Atomicity*, can accommodate pages collected at different points in time, as long as the procedure satisfies the causal relationships between memory operations and inter-process synchronization primitives. While this is a very elegant definition, it is unfortunately also extremely difficult to measure in practice, as it is almost impossible to enumerate all

causal relationships in a complex system. In fact, the same authors later estimated the atomicity of real dumps by simply computing the time delta between the first and last acquired pages [GF16]. Note that this is just an approximation, as even very short time delta may still result in snapshots that do not satisfy casual relationships.

What matters most is the fact that any kernel-level acquisitions are not atomic (neither according to the original nor the causal definition) [GF16]. To better examine the consequence of this fact, we need a more fine-grained measure of the discrepancy between different parts of a memory dump. For this reason, we introduce the concept of *time consistency*. A set of physical pages are *time-consistent* if exists an hypothetical atomic acquisition process that could have returned the same result or, in other words, if there was a point in time during the acquisition process in which the content of those pages co-existed in the memory of the system.

We argue that time-consistency is a more practical measure of the quality of a memory snapshot, as it can be accurately measured in an experiment (as we will describe in Section 3.4). Moreover, while it is still practically impossible to obtain a complete time-consistent image if the system is not frozen during the acquisition process, it is still useful to test this property on a smaller scale. For instance, an analyst may be interested in knowing if all the `EProcess` structures contained in a memory dump are time-consistent. In other words, even if the entire memory dump contains non-consistent data, individual processes or particular data structures may locally satisfy this property. Therefore, if we know that the acquisition of the process list is time-consistent, this may be enough to guarantee the correctness of a number of important memory forensic tasks.

3.3 IMPACT OF TIME IN MEMORY FORENSICS

The memory of a running system can be seen as a very large graph of interconnected objects. This is essentially what makes memory forensics possible in the first place. However, as we explained in the previous section, those objects are often collected at different points in time – with

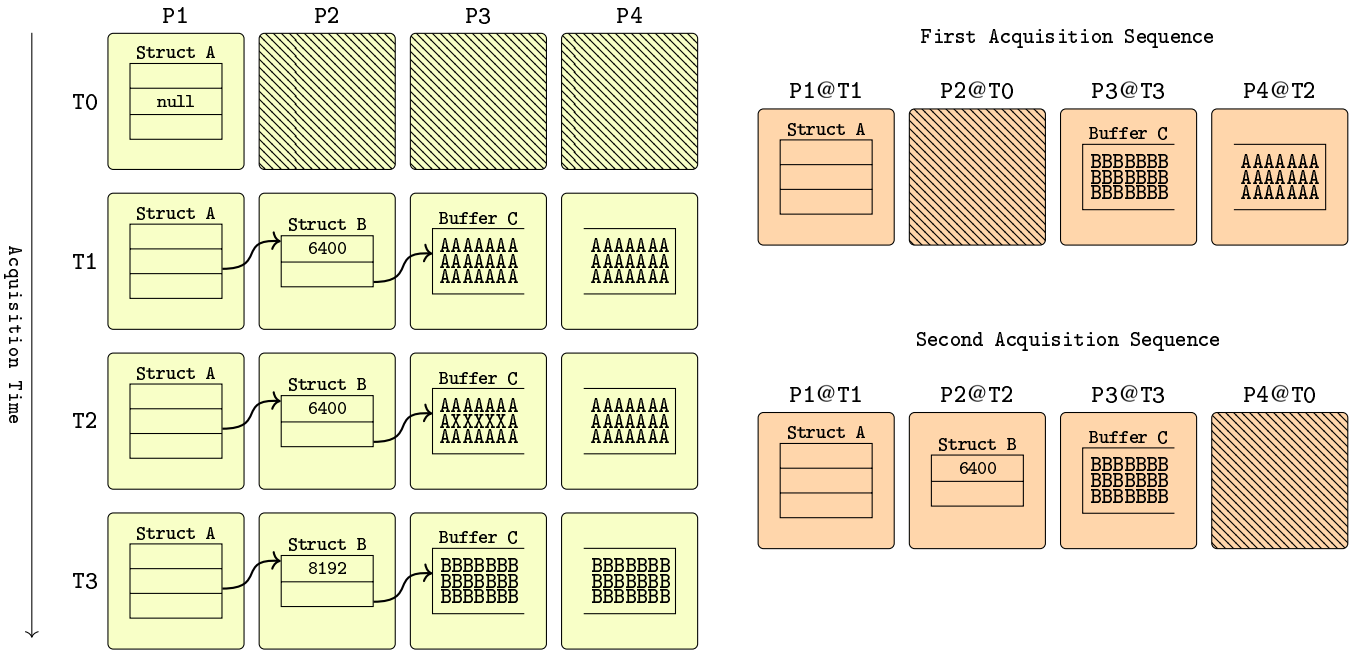


Figure 3.1: Example of Non-Atomic Acquisitions

the result that the values stored in different parts of the memory can be inconsistent and that pointers (i.e., the edges in the graph) can be incorrect and point to the wrong destination.

Figure 3.1 shows an example of the problems that can occur during a non-atomic acquisition. The left of the figure shows (in a simplified manner) three distinct but interconnected memory objects: a `Structure A`, which contains a pointer to a `Structure B`, which in turn points to a `Buffer C`. The two structures are stored in two separate physical pages (P1 and P2) while the buffer spans two pages (P3 and P4). All pages are represented sequentially along the X axis of the figure. The Y axis shows instead how the memory content evolved over time (represented in four discrete points: T_0, T_1, T_2, and T_3). As an example, we can imagine that at Time T_0 only `Struct A` is allocated. The other two objects are created at Time 1, when the buffer is also filled with 6400 characters (as indicated in the first field of `Struct B`). Part of the buffer is then modified at Time T_2, and finally its entire content is replaced by 8K new characters at Time T_3.

The right part of the figure shows two possible acquisition sequences for the four physical pages. We use the notation `Px@Ty` to indicate that Page X is acquired at time Y. For instance, the first dump corresponds to the acquisition sequence: $P_1@T_1$, $P_2@T_0$, $P_3@T_3$, $P_4@T_2$. We now use these two examples to introduce three types of inconsistencies that can affect a memory dump.

Fragment Inconsistency – This problem affects large objects that are fragmented over multiple physical pages, when their content (as acquired in the memory dump) is not time-consistent. For example, in the first acquisition in Figure 3.1, the buffer contains half of the content it contained at time T_3 and half of the one at time T_2. If this was data received over the network, such as a web page, a forensic investigation would find a mix of two consecutive messages, or a mix of the HTML code of two separate pages. While in our example we show a case of fragment inconsistency on a buffer, this can also affect large structures or arrays containing multiple elements.

Pointer Inconsistency – The second type of inconsistency affects the connection between two different objects, when the value of a pointer

and the data it points to are acquired at different moments in time. This is the case for `Struct A` in the first acquisition sequence. Its pointer to `Struct B` was acquired at time T2, while the content of the destination page was acquired at T1, before the second structure was even allocated. This can have serious and unpredictable effects on memory analysis. In fact, a forensic tool may try to follow the pointer and cast the destination bytes to a `Struct B` type. Sometimes the result can be easily identified as incorrect, but unfortunately arbitrary sequences of bytes can often be interpreted as valid data, thus leading to wrong results.

Value Inconsistency – The third and last type of inconsistency we present in this thesis occurs when the value of one object is not consistent with the content of one or more other objects. This is the case of P2 and P3 in the second acquisition sequence of Figure 3.1. The pointer in `Struct B` is not affected by the non atomic acquisition and it correctly points to `Buffer C`. However, the counter in the structure that keeps track of the number of characters currently stored in the buffer is not consistent with the content of the buffer itself (when its value was 6,400 the buffer did not contain any '`B`'). The page smearing phenomenon recently described by Case and Richard [CR17] is also an example of value inconsistency, in this case between the data stored in the page tables and the content of the corresponding physical pages.

As we already mentioned in the previous section, the fact that two pages are collected at a different point in time does not necessarily mean their content is inconsistent. For instance, P1 and P2 in the second dump of Figure 3.1 are time-consistent. Also pages P1 and P3 are time-consistent. However, all three of them together (P1, P2, and P3) are not.

3.4 IMPACT ESTIMATION

In this section we evaluate how often the inconsistencies presented in the previous section are present in a memory dump and what are the consequences on a number of common memory analysis tasks.

All experiments were conducted using a virtual machine equipped with 4 CPUs, 8GB of RAM and running Ubuntu 16.04 LTS. While the problem of non-atomic dumps is independent from the operating system, we decided to base our tests on Linux because the availability of its source code simplified the task of retrieving and double-checking the internal state of kernel objects. The physical memory was acquired using LiME [Syl12], a popular kernel-level acquisition module for Linux and Android systems.

We want to stress that while our experiments were conducted in a test environment, we strongly believe that this does not invalidate our findings. Moreover, the memory acquisition speed of our environment is comparable with the one exhibited by the fastest tool, as reported by McDown et al. [McD+16]. This puts our experiments in a best case scenario, and thus we believe the use of slower tools can increase the scope and the number of inconsistencies present in a non-atomic memory dump.

3.4.1 *Fragmentation*

In our first experiment we want to show how *fragmented* the physical address space is in an average computer. In fact, consecutive pages in the virtual address space of a process may be mapped to very distant pages in the system RAM. The actual location may depend on many factors, including the OS allocation policy, the number of running processes, the amount of available memory, and the uptime of the system (memory in a freshly booted computer is likely to be less fragmented than memory on a server that has not been restarted for months). For our test we took a conservative approach, using a Linux VM that had been running traditional desktop software (e.g., Firefox and VLC media player) for a period of only three hours. We expect a dump of a real system to be even more fragmented than what we measured in our test.

Figure 3.2 shows the physical pages assigned to a subset of the processes running in the system. The *X* axis shows the position (all pages were acquired sequentially) of the physical pages – where blue squares

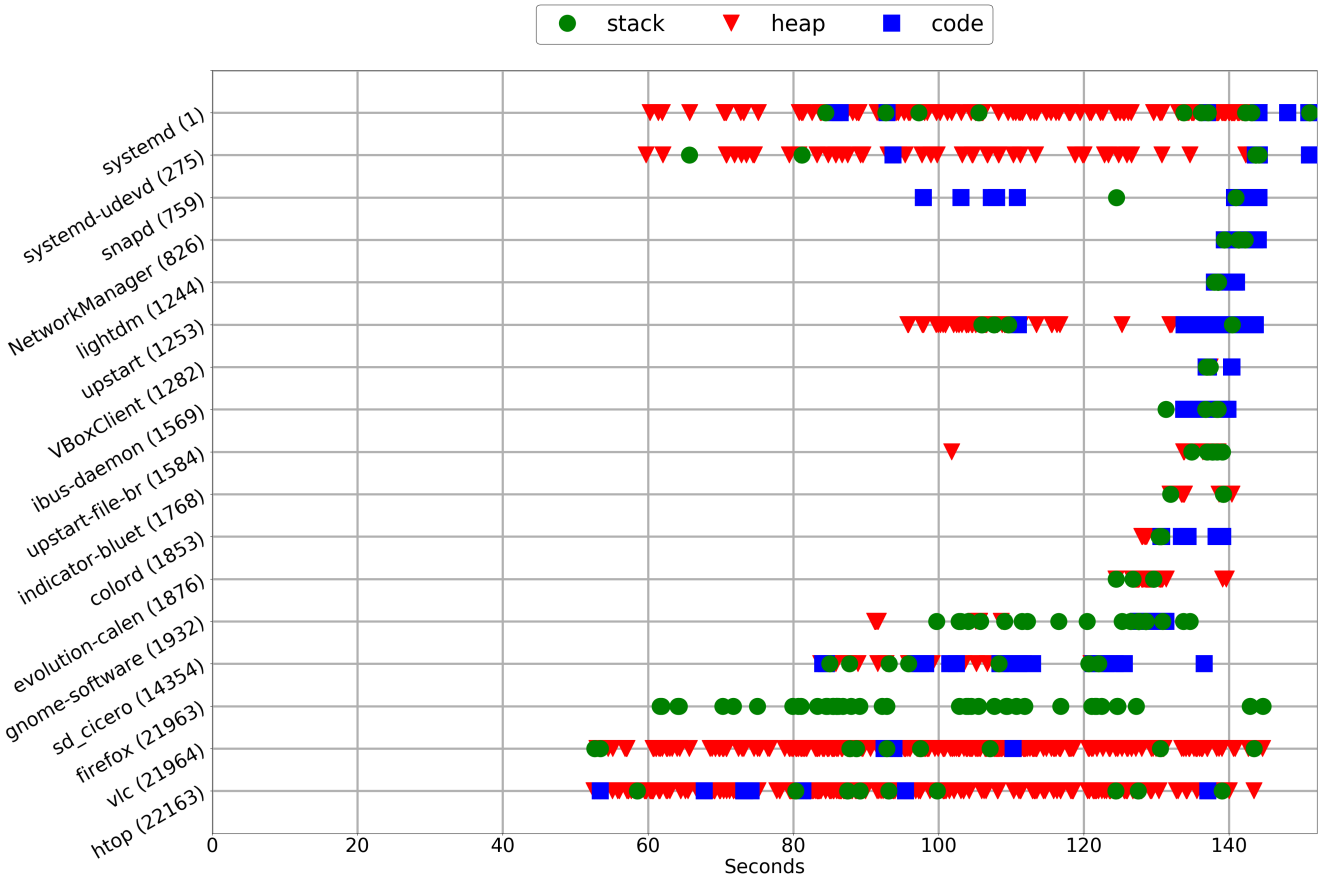


Figure 3.2: Example of physical memory fragmentation

represent program code and data, green circles the process stack, and red triangles heap pages. Since the tool acquired each page in the order in which they appear, the *X* axis also measures the time at which each page was acquired.

Thanks to this figure we can observe several interesting phenomena. First, for some processes (such as `NetworkManager`) all pages were located close to each other in the physical memory. Therefore, these pages were also acquired in a very short period of time. For other processes (e.g., `snapd` and `udevd`) the pages were instead scattered through the entire RAM, thus increasing their inter-acquisition time and the probability of containing inconsistent data. Another interesting case is given by the `Firefox` process, for which our analysis was unable to locate the physical pages containing the program code section. A closer analysis revealed that this was due to an *inconsistency* in the process VMA list (which, as we explain later in this section, Linux uses to store the memory region owned by a process).

To investigate this type of issues and better understand how the fragmentation may affect the results of an analysis, we performed two sets of experiments – one focusing on the integrity of kernel data structures, and one focusing on similar issues in the address space of a single process.

3.4.2 *Kernel-Space Integrity*

The OS kernel contains many useful pieces of information which are required during a forensics investigation. This information is spread over a multitude of interconnected data structures that evolve over time to keep track, for example, of resource allocation and the creation of new processes. These structures are connected by means of pointers in complex topologies such as linked lists, red-black trees, and graphs. A classic example of such topologies is the process list. In the Linux kernel every process is represented by a `task_struct`, which plays a central role in the kernel as it stores most of the information associated to each running process. At the time of writing, this structure contains more than 300 fields, and 90 of them are pointers to other structures. One of these

```
struct mm_struct {
  /* list of VMAs */
  struct vm_area_struct *mmap;
  /* RB tree of VMAs */
  struct rb_root mm_rb;
  ...
  /* number of VMAs */
  int map_count;
  ...
} ;
```

Figure 3.3: An excerpt of `mm_struct` taken from `linux/mm_types.h`

fields, called `tasks`, is particularly important because it is by following the pointers contained therein that a tool can enumerate the active processes in the system. Another valuable field part of the `tast_struct` is `mm`, which points to a structure (`mm_struct`) that contains all the information related to the virtual memory of the process, such as the first and last address of the stack and the heap, and a reference to every memory mapping requested by the userspace program.

Because of its central role in inspecting the memory of a process, we focused our evaluation on this structure. Page tables could have been another possible candidate for this test, as other researchers already noted inconsistencies in the page table when they are acquired non-atomically [CR17; GF16]. However, while this is known, the impact on other kernel data structures is still, as of today, unclear. Moreover, all modern operating systems implement *demand paging*, which means that user space pages are not loaded and mapped in RAM until they are needed – thus complicating the task of identifying possible inconsistencies. While this mechanism can be disabled by an application (i.e, using the `mlock` Linux system call) this would provide an unrealistic scenario to carry out our test on.

Figure 3.3 reports an excerpt of the `mm_struct` definition. The first important field is the `mmap` pointer, which points to a `vm_area_struct` (VMA). Each of these structures represent a different memory region

(such as a shared library, the code of the program, or its stack) and contain the information about the beginning and the end of a map, the permissions associated with it and, in case there is one, a pointer to the mapped file. This is the information that is used to populate the `maps` file under the `proc` filesystem.

A memory forensic analysis tool can reach these structures in two different ways. The first one is by traversing a linked list containing all `vm_area_struct`. The second is by traversing a red-black tree, which contains the same elements of the linked list and is rooted in the `mm_struct`'s `mm_rb` field. This redundant design allows the Linux kernel to search for a free area using the list, but also to take advantage of the tree topology to quickly check - for example during a page fault - if an address belongs or not to a VMA [Gor]. The last field that is important to understand for our evaluation is `map_count`, a counter that contains the number of VMAs associated to the process [2].

We now want to understand what happens to these data structures when a memory dump is collected in a non-atomic fashion. In fact, inconsistencies in these structures can result in serious problems during an investigation, as it would be impossible to know where the memory containing the program code, the shared libraries, the heap, and the stack of a process are located in memory. However, it is important to note that all the structures we mentioned so far are allocated using the `kmalloc` function – which ensures that two contiguous pages in the virtual address space are *also* contiguous in the physical memory. For this reason, even large structures cannot be affected by the *Fragment Inconsistency* problem we described in the previous section.

Experiments

To mimic different real world scenarios, we performed three experiments. In the first one we re-created a scenario where the investigator needs to

2 While these structures are specific to the Linux kernel, an equivalent tree is also present in the Windows operating system, under the name of Virtual Address Descriptor (VAD) tree. The use of the VAD tree in memory analysis has been previously described by Dolan-Gavitt [Dol07].

analyze the memory of a Firefox web browser that was left with five open tabs. The second scenario involves instead a potentially infected server machine, hosting a Wordpress installation, and handling a workload of 15 requests per second. In this case the examiner wants to analyze the content of the Apache memory, to understand if the server was compromised. The last scenario involves again a client machine, this time infected by a real malware.

For each scenario we collected 10 different memory dumps, which were then analyzed by a custom Volatility plugin designed to look for inconsistencies among the VMA-related data structures. In particular, for each process under analysis[3], it traverses both the list and the tree of VMAs, *counting* the number of elements they contain. It then compares this two numbers with the `map_count` field and prints an error message if the two values are different. The list exploration algorithm implemented in Volatility was left untouched. On the other hand, we had to fix the recursive exploration of the tree since the Volatility implementation was hanging on some memory dumps. We believe this is due to presence of malformed trees (a consequence of the non-atomic acquisition), for which the exploration was trapped inside an infinite loop.

The results for all the three scenarios are presented in Table 3.1. The first row of the table reports the fraction of processes for which the number of elements contained in the VMAs list and the `map_count` counter were inconsistent. Similarly, the second row contains the result of doing the same comparison, this time counting the number of elements in the red-black tree. The last row shows how many processes are affected by one or the other inconsistency. A stunning 78% to 100% of the analyzed processes in the three scenarios contained errors in their VMA information.

It is important to understand that a mismatch between the counter and the elements in the list (or in the tree) does not necessarily translates

3 In the malware and browser scenarios we analyzed only one process per memory dump. However, Apache in its default process management mode, spawns several processes to simultaneously handle all the incoming connections. Therefore, over the 10 dumps our tool analyzed the status of 153 `apache2` processes. For this reason, all results are reported as percentages.

Table 3.1: Experiment results

	Scenario 1 (Firefox)	Scenario 2 (Apache)	Scenario 3 (Malware)
List mismatch	100%	71%	80%
Tree mismatch	100%	73%	80%
Total	100%	78%	80%

into a serious problem or into wrong results, but it should nevertheless alarm the analyst. It is indeed possible that after the counter was acquired the process created a new memory mapping, thus increasing the number of elements in the data structures *without* seriously affecting the analysis of the process memory.

We therefore took a closer look at the elements present in those data structures, searching for some mappings that should always been present: namely the application *code* and its *stack* (we did not include the *heap* as programs like Firefox use their own custom dynamic allocator and therefore there is no corresponding entry in the VMAs structures). Surprisingly, in the Firefox experiment, the list of mappings *never* contained any entry for the application code nor for its stack. Being compiled as position independent code, the program is loaded into a fairly high part of the virtual address space. The VMA list is sorted according to the starting address of the memory area and Firefox had over 200 mapped entries which preceded the code. Therefore, any change in those entries can result in a corruption of the list that prevents the discovery of the area where the code is located. A closer look revealed that these mappings contained the custom heap of Firefox, which changes very rapidly – thus increasing the probability of an error in a non-atomic acquisition. The picture was only slightly better for the red-black tree, where the code

mapping was found three out of ten times, and the stack only once.

A similar result was found in the malware experiment. In this case the code and stack sections were missing in five out of eight inconsistent dumps according to the red-black tree. Using the list, it was instead always possible to retrieve the code entry, while the stack was missing in three out of eight experiments.

Atomicity and Quiescence

Even a perfectly atomic dump acquired on a frozen OS can contain errors in some data structures, if the acquisition was performed while the operating system code was in the middle of a data structure update [Hof+11] (e.g., while adding or removing an element from a list). Therefore, one may argue if the inconsistencies we detected in our experiments are due to the lack of time-consistency or to the fact that the image was taken while the OS was not in a quiescence (i.e., idle) state.

While it is not possible to distinguish the two cases, we are confident that time was the culprit of the errors we detected for two important reasons. First, because while the OS can indeed be in the middle of updating some pointers, this can only affect a very limited group of data structures in a dump (i.e., the OS cannot be in the process of updating the memory maps of *all* running processes). Second, when we repeat the same three experiments using a different acquisition strategy that can preserve the atomicity of certain memory regions (see Section 3.5 for more details), all the inconsistencies we reported in the previous experiments disappear.

Impact

To conclude we want to emphasize why this seemingly insignificant problem can have a serious impact on an investigation. First of all, inspecting the memory of a malicious process, or of a potentially compromised application, is a common task in memory forensics. If the analyst cannot trust the VMA information, important areas of memory can be missed or wrongly attributed to a different process. On a different example, few days after the spread of the `WannaCry` ransomware attack, a decryption

tool (`wannakey` [Gui17]) was published by a security researcher. The tool works by extracting from the malware memory the prime number of the private `RSA` key used by the ransomware to encrypt the victim files. In a similar attempt, a Volatility plugin [Mar17] searches for the `AES` keys used by the `NotPetya` malware. This shows how memory analysis often relies on locating and extracting a small amount of data from a process memory. But our experiment highlights that the apparently trivial task of retrieving all memory regions mapped by a process results in wrong information in *over* 80% of the dumps acquired with a non-atomic solution.

3.4.3 *User-Space Integrity*

We now shift our focus to userspace applications, to understand if they are also impacted by similar problems. Most of the data from a running application can be retrieved by looking at its stack and heap segments. The heap contains dynamically allocated objects, which are often aggregated in complex high level structures such as lists and trees. Therefore, the type of errors introduced by inconsistent pages is similar to what we already discussed in the kernel-level experiments. On the other hand, the stack has a special role in the process execution, as it is responsible for maintaining information about the current state of nested functions invocations, along with the values of their parameters and local variables. This is very important to understand what the program was doing at the time the memory was acquired, and to extract the pointers to dynamic objects required to initiate a heap analysis. For this reason, in our second set of experiments we decided to look at the consequences of non-atomic dumps on the analysis of the stack of a program.

To perform these tests, we need to be able to reliably reconstruct the stack trace of a process starting from its memory dump. This task may be complicated by a number of different factors (e.g., library functions that do not use the frame pointer, or lack of symbols information). However, since our goal is to investigate inconsistencies in the stack pages, without loss of generality we can prepare the environment to facilitate this analysis. In

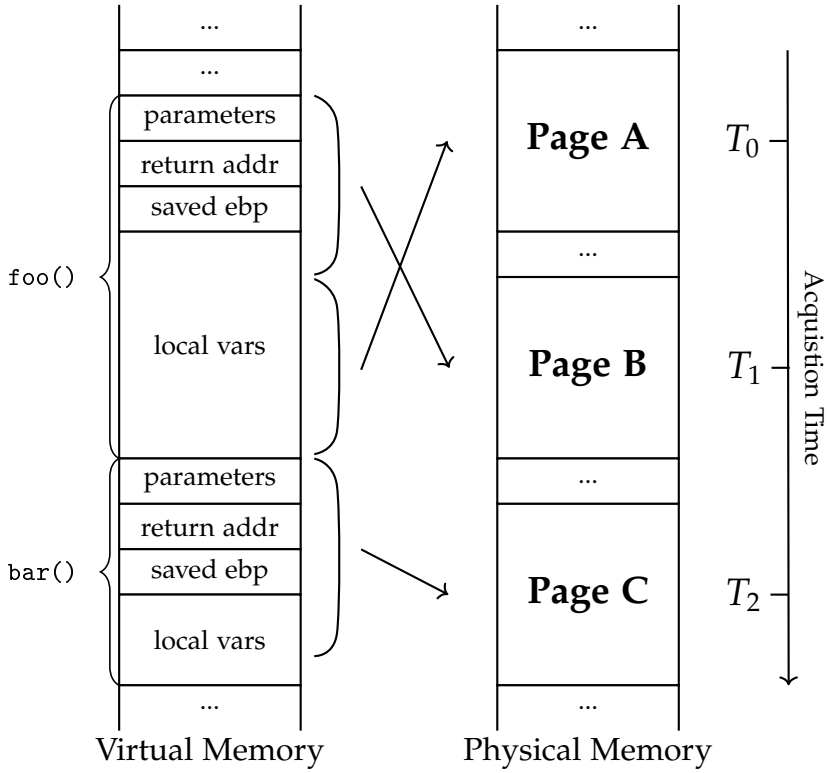


Figure 3.4: Stack layout of the target program. On the left a view from the virtual memory perspective, on the right from the physical one

particular, we compiled the target application (`bzip2`) and all its libraries without any optimization (`-O0`). We then disabled the stripping of debug information, to retain the function names and corresponding addresses in the binary.

Figure 3.4 shows a portion of a stack trace. The initial values of `rip` and `rbp` are obtained from the *kernel stack* of the process, reachable from the process's `task_struct`. This is where the kernel saves the content of all registers upon a context switch. Using this information, we can locate the current function and then walk back to the previous frames by following the `saved rbp` field. The left side of the figure shows the

activation records of two hypothetical functions (`foo()` and `bar()`). It is important to note that while the two frames are always contiguous in the virtual memory of the process, the physical pages that contain them can be quite far apart, and even appear in a different order (right side of Figure 3.4). Moreover, the frame of `foo()` in our example spans across two separate memory pages, thus fragmenting its content in non-adjacent physical addresses. Finally, the figure also shows the temporal axis that marks when each part of the memory was collected by a hypothetical acquisition process. In our case, the first part of `foo()` activation record was dumped at time T_1 and its second half at time T_0.

All three types of inconsistency introduced in Section 3.3 are potentially present in our example: fragment inconsistency in the local variables of `foo()`, value inconsistency among the content of the two frames, and pointer inconsistency between the `saved rbp` of `bar()` and the content of `Page B`. Moreover, the kernel stack of the process - which is where the initial value of `rbp` and `rip` are extracted from – also lives in another memory page which could potentially be collected long after or long before T_2. In this section we will therefore try to measure how the lack of atomicity in the acquisition of the userspace and kernelspace stacks affects the process of creating a correct backtrace of a running application.

Experiments

To identify which errors are introduced by the non-atomic acquisition we need a ground truth to compare with. We solve this problem by modifying the LiME tool to keep track of changes on the stack content during the acquisition time. In particular, we modified LiME to save an atomic view of the target process' stack *any time* one stack page was acquired by the normal acquisition process. In our example, this resulted in acquiring the content of the three pages (`Page A`, `B`, and `C`) at T_0, T_1, and T_2. The atomic view also included the value of `rbp` and `rip` taken from the kernel stack. To rule out the possibility that our kernel module and the target program run at the same time on two different CPUs, therefore invalidating the registers saved in the kernel stack, we executed this experiment using a VM with a single CPU. Also, to ensure our

Table 3.2: Experiment results

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}
Frames	-	6	-	6	8	-	-	-	6	-
Physical Pages	4	5	5	4	4	5	4	4	5	5
Acquisition Time (s)	3.2	30.0	37.8	37.0	0.25	26.0	28.6	1.0	27.6	39.9
`rbp` delta (s)	7.7	38.8	49.6	43.7	7.3	43.4	4.3	4.0	15.1	5.64
Corrupted (`registers`)	✓	–	✓	–	–	✓	✓	✓	–	✓
Corrupted (`frame pointers`)	–	–	–	–	–	–	✓	–	–	–
Inconsistent data	N/A	✓	N/A	✓	–	N/A	N/A	N/A	✓	N/A

kernel module does not get preempted by other code with higher priority while is acquiring the atomic view, we used the `lowlatency` version of the Ubuntu kernel, which is compiled with the `CONFIG_PREEMPT` kernel option. This enables two additional macros that allow our code to disable preemption before acquiring the required elements and to re-enable it afterwards. These macros are enough to protect a small critical section but our module calls complex kernel functions that may "intentionally" call the `schedule()` function to pass control to another task. When this happens, the kernel produces a warning message that we can monitor to discard the image and repeat the test until no warning message is generated.

This complex set up was required to be able to reliably monitor what happens to the process stack during the memory acquisition process. Therefore, we can expect that in a real setting with multiple CPU cores executing concurrent code, the number of inconsistencies would be substantially higher.

We repeated the experiment twenty times, ten on an idle system and ten on a system under a high workload. Since we did not find any significant difference among the two sets, we present in Table 3.2 only the result for the idle system. Each column in the table represents a different memory dump. The first four rows show some statistics about the stack trace of the target process, including the number of identified frames and the number of stack pages effectively used by the application. The third row reports the time difference (in seconds) between the first and the last acquired page of the stack, while the fourth row contains the difference between the kernel stack acquisition time and the time the page containing the top stack frame was acquired. Intuitively, the larger the two time windows the more likely the process was to modify the stack during the acquisition, resulting in possible incorrect results for the analysis.

The bottom half of Table 3.2 reports the problems we run into when running our Volatility plugin to extract the stack trace on each memory dump. The table lists three separate cases. In the first two, marked as *Corrupted*, our plugin was unable to generate a complete stack trace. This

was due to two different reasons. The first (marked as `registers` in the table) is an inconsistency between the registers stored in the kernel stack and the state of application stack, which caused `rbp` to point to an incorrect location. This was the case for six out of ten dumps. However, one may argue that an analyst may resort to other heuristics to locate the top frame on the stack, without the need to recover the registers from the kernel. Therefore, we re-run our Volatility plugin by providing the initial correct value for `rbp`, and therefore the position of the top frame, as a parameter. The next row in the table (marked as `frame pointer`) reports the cases in which, despite the top frame being identified manually, it was still impossible to retrieve the entire stack trace. In this case the problem was that adjacent stack frames resided in *non-adjacent* physical pages acquired at different points in time. As a consequence, the saved `rbp` of one function pointed to a page that had been already overwritten with other data.

For the four dumps for which it was possible to reconstruct the stack trace, the last line in Table 3.2 shows the images in which at least one stack frame spanned multiple physical pages containing inconsistent information. This happened in three out of four cases (marked as `Inconsistent data` in the result table).

To conclude, only in one out of 10 dumps it was possible to retrieve a correct backtrace that was neither corrupted nor inconsistent. In other words, it seems that errors introduced by non-atomic dumps *are the rule and not the exception*. Not surprisingly, the only correct case corresponded to the acquisition in which all the stack pages were collected in less than 250 milliseconds. This, as we better explain in the Section 3.5, prompted us to look for possible solutions to minimize the acquisition time windows.

Impact

Researchers have recently proposed several forensic analysis techniques tailored for user space applications [Ots+18; BD17; AD07; CR16; Mac13]. Despite this effort, forensic tools still support only a *"small percentage of applications that hold forensic value"* [CR17] and experts are thus urging

the community to extend existing tools to support web browsers, office applications, and web and database servers. While the methodology used to locate and extract information from these applications may be very different, they all share a common unwritten assumption: they *require* a consistent view over the data of a process, whether it is stored on the stack or on the heap. We believe that our experiments draw attention to a problem too often overlooked by the forensic community, and will help researcher to better assess their methodologies in the presence of non-atomic acquisitions.

3.5 A NEW TEMPORAL DIMENSION

The importance of the temporal dimension has been largely underestimated in memory forensics. In the previous sections we showed that this negligence can lead to wrong results during several memory analysis tasks. While our experiments shed light on an important problem, we did not discuss possible ways to mitigate the problem.

In this section we tackle the problem by following two different approaches. First, we discuss how we can record precise timing information during the memory acquisition phase and integrate this information in the forensic analysis process. Second, we discuss how the lack of atomicity can be mitigated, by performing a non-linear acquisition scheme.

3.5.1 *Recording Time*

Given an object and its address in memory, there should be a way for the analyst to know the exact moment in time in which that part of the memory was acquired. This gives the forensics analyst the ability to estimate the degree of atomicity among two or more objects used during an analysis task. To record this information in the first place we modified LiME [Syl12] to produce, along with the memory content, a log file containing precise time information about the acquisition process. Our implementation allows the user to save a timestamp for each page,

or to only output the time information at particular intervals (e.g., once every 10ms).

We then conducted three experiments on a bare metal machine equipped with 8GB of RAM: a baseline test without any time acquisition, one test that saved a timestamp every $100\mu s$, and one that logged the actual acquisition time for every page. The baseline acquisition was completed in 83.92 seconds. The second experiment logged over 85,000 timestamps with an almost negligible overhead of 0.6 seconds (0.7%). Finally, the third experiment logged over 2M timestamps with 1.98 seconds (2.4%) overhead. Since any extra time spent to record the dump increases the likelihood that kernel and application data were modified during the process, our approach let the user choose the balance between the extra acquisition time and the precision of the logged timestamps.

At this point one may argue that a much simpler and more efficient solution exists to the time recording problem. It is in fact possible to simply take the total time spent to acquire the entire memory, assume it was uniformly distributed among all pages, and divide it by the number of physical pages to compute the time required to dump a single page. Given this number, it is trivial to label each physical page with an *estimation* of its acquisition time. This solution is appealing at first glance as it does not require changing existing tools and does not introduce any overhead in the acquisition process. However, we observed that in reality the acquisition code does not always run at the same pace from the beginning to the end of the dump. This is due to variables related to the state of the system that are impossible to evaluate *a posteriori*, such as the status of disk buffers and caches or the fact that other programs may suddenly require more CPU or may start writing to the disk, thus affecting the acquisition speed. To verify what is the error introduced by using this naive approach, we compared the estimated timestamp with the real one collected using our version of LiME configured to log the time for every page. Obviously, the *skew* between the timestamps of the first and last page is zero. For the remaining pages, the average skew was 4.4 seconds, with a maximum error of 8 seconds – which is unfortunately way too high to properly perform any time-based analysis. The modified LiME software can instead achieve very high precision (less than 1ms

error) with less than 1s overhead to acquire 8GB of RAM – which we believe is perfect to be integrated into production systems.

3.5.2 *Time Analysis*

We now look at how we can integrate in Volatility the time information we recorded. Our goal is to add this new dimension transparently, in order to be able to enrich the output of every command and plugin that is already available for the framework.

Volatility internally represents every OS structure with a dedicated standalone object, which contains different information such as its name, the data type of its fields, and the memory address where the structure is allocated. We modified the constructor of this object's class to retrieve and store the timestamp associated to the physical pages where each structure is stored. As Volatility reads some memory regions without passing through one of the structure objects (for example when it walks the page table), we also patched the code responsible to handle direct reads. All timestamps are then recorded inside an *access timeline* which is printed, along with other statistical information, after any Volatility command is completed. Moreover, the timing API we developed are exported to plugin developers, so other code can easily perform queries to know when a particular structure was acquired. Our current prototype presents to the investigator the number of physical pages accessed by the analysis, the time window in which those pages were originally acquired, and a simple graphic representation of the *access timeline*. A new parameter, `pagetime`, enables the timeline tracking and the display of this summary. Here is an example of the output for the `pslist` plugin:

```
$ ./vol.py -f dump.raw --profile=... --pagetime pslist
<original pslist output>

Accessed physical pages: 171
Acquisition time window: 72s
[XX------------XxX---xXXX--xX-xX---Xxx-xx-X-XxxX-XXX]
```

Table 3.3: Selected subset of Volatility plugins

Name	Description	Acquisition Timeline Bar
`bash`	Recover bash history	`[xX--------------------XXXXXXX----XXX---X--X---XX--X-XX---XX-XXXXXXXX-XXXX-]`
`dmesg`	Gather dmesg buffer	`[X-----------------------------------------------------------------------]`
`dump_map`	Dump mappings to disk	`[XX-----------------XxXXXXXxX---xXXX--xX-xX-xxxXxxxxxX--xxX-XXXXXXXxxXXXXX]`
`elfs`	Find ELF binaries	`[XX-----------------------X-----XXXX---X--X----X-X--------X-X-XXXXX---XXXX]`
`ifconfig`	Gathers active interfaces	`[X-----------------------------------------------------------------X----XX-]`
`info_regs`	Print kernel stack registers	`[xX-------------------------------x-------X--------------X-X-XXXXX----XX-]`
`lsmod`	Gather loaded kernel modules	`[x-----------------------------------------------------------X--------X--]`
`lsof`	Lists file descriptors	`[XX----------------------------X----------X------------X--X-X-XXXXX----XX-]`
`proc_maps`	Gathers process memory maps	`[xX------------------------X--------------X--------------X-X-XXXXX----XX-]`
`psaux`	Gathers process command line	`[xX---------------x---xxxxxXxXxxXxXXxxxx-x-Xxx-XxxxxxxxxxxXxXxXXXXXXXxXXXx]`

This information provides a first insight into the level of atomicity of the data structures used during a given task. If they are very dispersed over the memory it means that Volatility traversed structures that were taken far away in term of acquisition time. On the other hand, a narrow acquisition time window means that the information required to run the task is less likely to contain inconsistencies and the final result is therefore more reliable.

As an example, Table 3.3 shows a subset of common Volatility plugins, alongside a short description and the barplot of the physical pages used by the command when executed on our 8GB test machine[4]. We selected these particular commands as they traverse a different set of data structures, related to processes, files, network, and memory management. As expected, the outcome of this experiment is quite worrying. Only one plugin out of ten (`dmesg`) is using structures collected close in time. The rest relies on structures spread over the entire memory and therefore collected tens of seconds apart.

Note that our timeline can only *suggest* the presence of inconsistencies, pointing the analyst to result that might be incorrect and that therefore should require extra validation. Unfortunately, it is not possible to detect the *actual presence* of inconsistencies, since Volatility has only access to one dump and has therefore no ground-truth information to compare it with. However, it is possible to modify individual plugins to detect and report particular cases of *Value Inconsistencies*, such as the difference between the counter of VMAs associated to a process and the number of entries in the VMA list discussed in Section 3.4. This information could be used to print more fine-grained warning messages to further alert the analyst that particular care must be put in validating the obtained results. However, this process would require manually changing each individual plugin based on the actual semantics of the information it processes, which is outside the scope of our paper. Nevertheless, the presence of our temporal API allows plugin developers to move in this direction.

4 Note that all bar plots have a `x` in the first position. This is due to an access at the beginning of the physical memory, that contains the data section of the kernel. This section contains the `init_task` variable which is used by almost every plugin as an entry point for the process list.

3.5.3 *Locality-Based Acquisition*

So far we discussed how we can enrich a memory dump with timing information and how this information can be integrated in a memory forensics analysis tool. We now investigate if it is possible to mitigate the side effects of the lack of atomicity and reduce the serious inconsistencies depicted in Section 3.4.

As already explained in Section 3.2, the normal approach adopted by LiME and other memory acquisition tools is to scan the memory from the lowest up to the highest physical address assigned to system RAM and save the content of each page *sequentially*. This simple approach treats each memory page as equally important, whether or not it is actually used by the system and independently from the fact that it may contain crucial forensic information.

We propose instead a different approach that maximizes the "local atomicity" by acquiring the memory in consecutive chunks based not on their position but on their content. The idea is that pages that contain interconnected data (e.g., the element of a linked list) should be acquired in a short amount of time to minimize the chance of errors. Our acquisition algorithm is divided in two phases. The first, which we dubbed as *smart dump*, uses OS information to locate and dump a number of physical pages. For instance, it groups together pages that hosts popular forensic-related information such as the list of processes and the list of loaded kernel modules. Moreover, for each process it dumps its page table, the `cred` structure (which contains the security context of the process such as the `UID` and the `GID`), the list of memory mappings and the content of the heap and stack segments, the list of open files and the related structures, and the kernel stack of the process which contains the value of its registers after the last context switch. This ensures that while two different applications can be acquired far apart in time, the data of a single application is dumped in the minimum amount of time possible.

The second part of the acquisition consist of the traditional sequential dump of the remaining physical pages. To avoid the overhead of acquiring the same page twice, the smart dump keeps track of the acquired pages

in a bitmap. In this way, no single page of memory is acquired more than once. Our experiments show that the additional logic in the smart dump and the bitmap management add a *negligible* overhead to the acquisition time. Moreover, the kernel module is only 15 kilobytes larger than the one used by the traditional linear version. On the memory footprint side, the biggest object allocated by our module is the bitmap. For 8GB of RAM it requires 256KB of memory, which are nevertheless allocated in an area reserved only for kernel modules. Moreover, given that our solution does not need to allocate specific kernel structures, it does not have any impact on kernel caches - which often contains valuable information [Cas+10].

Note that our acquisition sequence relies on information provided by the OS, which are under control of the attacker in a compromised system. For instance, a rootkit can hide an entire process, which therefore would not be acquired during the smart dump phase. But even in a compromised system, our smart dump does not produce worse results compared with a traditional linear acquisition. In fact, since all the other processes are acquired in a local-sensitive way, this forces the remaining pages (including the hypothetical malicious process) to be also acquired in a shorter time window during the linear phase.

The same effect is obtained if a malware tries to intentionally increase its memory footprint to force our acquisition algorithm to spend more time acquiring useless pages. Again, this would force the system to postpone the acquisition of other kernel data structures in the remaining time (as the total acquisition time is constant and cannot be manipulated by a malicious actor) – thus resulting in a more consistent dump overall. The only successful strategy would be to increase the memory footprint and position key pointers far away from their targets. This would result in possible inconsistencies using our solution, but certainly not worse than what would be experienced with existing approaches.

Finally, we want to stress that our locality-based acquisition does not require any additional information that is not already used by linear-based approaches – i.e., the kernel headers.

To test if this solution would be sufficient to avoid the problems we detected in the experiments presented in Section 3.4, we re-run all user-

and kernel-space integrity tests on memory dumps acquired using our locality-based approach. In *all* cases we were able to retrieve a correct stack trace, thanks to the fact that the physical pages containing the process and kernel stacks were all acquired in less than 40 milliseconds. Similarly, the VMAs list and tree structures were also collected together, resulting in no inconsistencies and no missing mapped sections. These results apply to any other analysis that relies on information extracted from the stack or heap of a process, or on the content of the kernel data structure currently supported by our smart dump phase.

3.6 DISCUSSION

While the analysis and the results presented in this chapter are limited to few selected data structures, it would be extremely difficult to do otherwise. In fact, extending the test to all data structures is infeasible as a running Linux kernel includes thousands of different structure types. Moreover, to detect the presence of a corrupted information we need to compare against the same information taken from a different source. For instance, in one of our examples, we compare the VMA counter, the number of elements present in the list, and the one in the red-black tree. Unfortunately, this "ground truth" is not always available and ad hoc rules and experiments would be needed to support each individual structure. However, since there is nothing special about the VMA structure and since the lack of atomicity affects equally all memory, there is no reason to believe that inconsistencies would not affect other data structures managed by the kernel. We actually believe that our experiments were very conservative, and a real running system would suffer even more from this problem.

It is important to stress the fact that the goal of this paper is not solve the problem of lack of atomicity in memory dumps. It is instead to draw attention to this important and overlooked problem and show that it is not simply something we need to accept as a factor out of our control. Our contribution is to provide the tools to make the lack of atomicity evident to the analyst so that *time* becomes an aspect that can

be measured and taken into account in an analysis. Second, we show that a linear acquisition, as implemented by all tools on the market, is not an optimal solution and better approaches can be easily implemented to reduce errors due to the passing of time. Therefore, more research and more papers are certainly needed to improve the forensic field, not just from its *space* but also from its *time* dimension.

4 TOWARDS AUTOMATED PROFILE GENERATION FOR MEMORY FORENSICS

4.1 INTRODUCTION

The main focus of memory forensics is to parse and extract evidences from the data structures used by the operating system kernel. While some of these structures can be located by carving the memory for particular byte patterns, the true power of memory forensics comes from the so-called structured analysis. In most of the cases, this type of analysis starts by finding a set of global symbols inside a memory dump. From these variables, other kernel structures are then discovered by de-referencing pointers. For example, a common task performed in memory forensics consists of listing the processes that were running inside the machine when the memory dump was acquired. Under Linux, a way to retrieve this information is to find the location of the global variable `init_task` and use it to traverse the list of `task_structs`. However, even this simple and seemingly straightforward operation can be performed only if the tool has a *very* detailed model of the system under analysis. In memory forensics, this detailed model is called a *profile*. A typical profile contains two different pieces of information: the address of kernel global variables and the layout of kernel objects. The latter is of particular interest because it is influenced by several different factors, including the kernel version, the way the kernel was configured at compile time, and the compiler optimizations. Without a complete profile, none of the existing memory forensics tool is able to analyze a memory dump [CR17].

For Microsoft Windows operating systems, retrieving the correct profile from the system under analysis is not really a problem, because the number of different kernels is limited and well known. Moreover the layout can be retrieved from the kernel debugging symbols, which are generally available on a public symbols server. On the other hand, memory forensics is more and more focusing on Linux-based operating

systems, both for the analysis of servers infrastructure and to support a wide range of appliances, mobile phone, and network devices. Unfortunately, when it comes to Linux there is no central symbol server and the number of combinations of kernel versions and possible configuration is countless. Thus, forensics analysts have to manually create a profile for each and every system they want to analyze. Currently, this is a manual process that has several implicit requirements. For instance, it requires a compiler toolchain and the kernel headers to be installed on the system under analysis. While certain desktop machines might satisfy the previous constraints, this is seldom the case for `IoT` devices, smartphones, or servers – thus effectively limiting the applicability of memory forensics. Moreover, this process involves the compilation of a kernel module, out of which the layout is extracted. This is indeed a problem because the module needs to be manually updated every time a new kernel is released, because not all the structures used by memory forensics tools are exported in header files. Finally, a new serious "threat" to memory forensics is arising: *structure layout randomization*. Originally implemented by Grsecurity [Spe06], layout randomization is nowadays present in the latest versions of the Linux kernel as well. This compile-time option randomizes the layout of sensitive kernel structures, as an effective way to harden the kernel against exploitation. By their own admission, the authors say that enabling this option might "prevent the use of forensic tools like Volatility against the system".

The memory forensics community is well-aware of all of these problems, as recently emphasized once again by Case and Richard [CR17] in their overview of memory forensics open challenges. In the paper, the authors urge the community to create a public database of Linux profiles - which nowadays exists only in the form of a commercial solution. Unfortunately, they also note how a "considerable amount of monetary and infrastructure" is needed to create such a database and how, in any case, this approach can only cover settings used by pre-compiled kernel shipped as part of Linux distributions.

In the past years, researchers have also proposed partial solutions to the profile generation problem. For example, Case et al. [CMR10] and

subsequently Zhang et al. [ZMW16], suggested that the layout can be retrieved from the analysis of kernel code. Unfortunately, their *manual* approach covers only an handful of kernel structures layout, while nowadays memory forensics requires several hundreds of them. On the other hand, approaches such as the proposed one by Socała and Cohen [SC16] still requires the configuration that was used to build the kernel under analysis. For these reasons, in this thesis we propose a novel approach to automatically create a Linux profile, using *only* information publicly available or extracted from the memory dump itself.

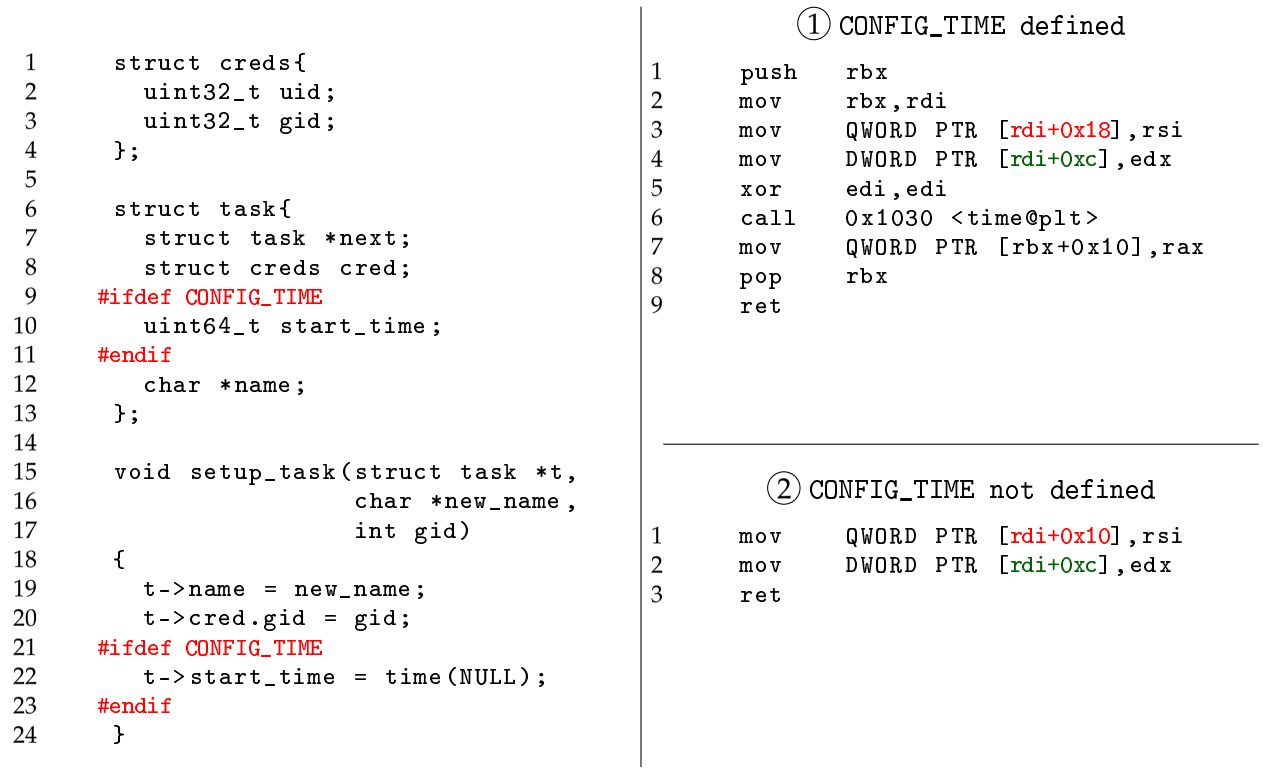
Our experimental results show how the profile extracted by our framework supports several Volatility plugins - such as those that list the processes and the open files - when targeting a very diverse set of kernels. This set includes a version of the kernel currently shipped by Debian – used once with structure layout randomization and once without – an Android kernel, a kernel shipped by Operwrt (a project targeting network devices) and an version of the Ubuntu kernel released 8 years ago.

4.2 RECOVERING OBJECTS LAYOUT FROM BINARY CODE

In this section we discuss a practical example of how the layout of an object is shaped by the configuration used at compile time, thus making it impossible to deduce the offsets of its fields by reasoning only on its definition. We then introduce the core idea behind this paper and how it can be generalized to recover the layout of all kernel objects used in memory forensics.

4.2.1 *Problem Statement*

The key ingredient that makes memory forensics possible is the availability of the kernel *profile*: a detailed model of the symbols and data types required to perform the analysis. In the case of Linux memory forensics, a profile contains two separate pieces of information: the addresses of global variables and kernel functions, and the exact layout of kernel objects. The latter is of particular interest for different reasons. First of all,



```
1     struct creds{
2       uint32_t uid;
3       uint32_t gid;
4     };
5
6     struct task{
7       struct task *next;
8       struct creds cred;
9   #ifdef CONFIG_TIME
10      uint64_t start_time;
11  #endif
12      char *name;
13    };
14
15    void setup_task(struct task *t,
16                    char *new_name,
17                    int gid)
18    {
19      t->name = new_name;
20      t->cred.gid = gid;
21  #ifdef CONFIG_TIME
22      t->start_time = time(NULL);
23  #endif
24    }
```

① CONFIG_TIME defined

```
1     push   rbx
2     mov    rbx,rdi
3     mov    QWORD PTR [rdi+0x18],rsi
4     mov    DWORD PTR [rdi+0xc],edx
5     xor    edi,edi
6     call   0x1030 <time@plt>
7     mov    QWORD PTR [rbx+0x10],rax
8     pop    rbx
9     ret
```

② CONFIG_TIME not defined

```
1     mov    QWORD PTR [rdi+0x10],rsi
2     mov    DWORD PTR [rdi+0xc],edx
3     ret
```

Figure 4.1: On the left the C source code we use in our examples, on the right its compiled form

this information is lost during the compilation process and the only way to preserve it is to ask the compiler to emit the debugging symbols. This is often the case for kernels shipped by common Linux distributions – usually distributed in a separate debugging package. Moreover, the Linux kernel is a highly customizable piece of software, designed to run on a large variety of devices and architectures and to suits different needs. This means that the very same kernel version tailored to two different systems can result in dramatic differences between the layout of the kernel objects.

To illustrate how the customization of the Linux kernel is in fact a problem for memory forensics, we present a practical example in Figure 4.1. In the left part of this figure we show a short code snippet responsible for the set up of a task which, in this example, is represented by the `task` object. Every task has a pointer to the next task, some credentials, and a name. Moreover, in case the macro `CONFIG_TIME` was defined at compile time, a task also includes the field `start_time`. The function `setup_task` initializes a task and its fields. In the right part of the Figure we instead report the disassembly of two versions of this function, one in which the macro was defined at compile time ①, and one in which it was not ②.

The first difference between the two versions is present at lines 3 and 1, respectively. The semantic of this two instructions is equivalent: they store the argument `new_name` (passed in the `rsi` register) into the `name` field. However, the offset of this field is different between the two version, and so is the displacement from `rdi` (which contains the `t` argument). This is a consequence of the fact that in ① the compiler had to reserve 8 bytes before the field `name` for the `start_time` field, while in ② the latter was entirely removed by the preprocessor. On the other hand, the same displacement is used to access the field `gid` of `creds` at lines 4 and 2. This is because the field `cred` - and subsequently the field `gid` therein contained - precedes `start_time` and thus is not concerned by its presence or by its absence.

While this is a trivial example, it introduces a very common pattern that is present thousands of times in the kernel codebase. For example, the definition of the `task_struct` alone - which is one of the most important object in memory forensics - is shaped by more than 60 different #*ifdef*s.

The large number of combinations that derive from these definitions make it impractical to enumerate all possible offsets where a field can be located. However, as we saw in our little example, this information is encoded into the compiled code and therefore we believe the only practical way to automatically recover the layout of kernel objects is by *extracting* it from the kernel binary itself.

4.2.2 *Data Structure Layout Recovery*

The intuition behind this paper is that, while the exact fields location is lost during the compilation process, it affects the code generated by the compiler. More specifically, the displacement used to access the fields of a given object must reflect the layout of the data structures and therefore can be extracted if we know *where* each field is used across the entire codebase, and *how* the code is accessing the field. These two pieces of information allow us to locate the functions that operate on the requested field, and to follow the access pattern that led the code to a particular object. For example, a piece of data can be passed as parameter, but it can also being referenced by a global variable, reached by traversing another object, or obtained by calling a separate function.

Back to our example, let's assume we want to recover the offset of the *name* field. First, by looking at the source code, we can tell that the function `setup_task` accesses this field and also that the variable `t` is passed as parameter. Given that the Abstract Binary Interface (ABI) of `x86-64` [Mat+13] specifies that the first parameter is passed using the `rdi` register, we can perform a data-flow analysis and track every memory access whose value depends on the `rdi` register. In version ①, this happens at lines 3 and 4, but also at line 7 because `rbx` was initialized from `rdi`.

It is important to note that it is very difficult to tell which of the three access is the one operating on the field we are interested in. In fact, functions often access dozens of different fields and compilers optimizations often change the exact order and number of those accesses in the binary code. However, we can leverage the fact that the *name* field is also

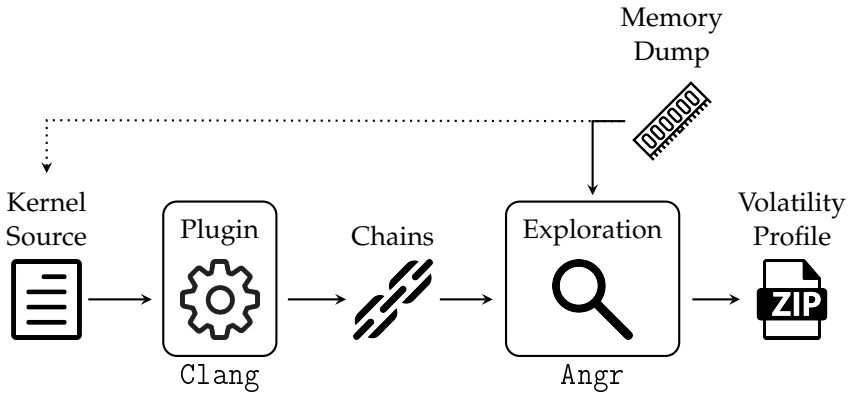


Figure 4.2: System Overview.

probably accessed in other functions, and therefore we can combine and cross-reference multiple candidate locations to narrow down its exact offset. In Section 4.6.1 we will describe in more details the numerous challenges the layout recovery algorithm needs to face when dealing with complex kernel code and the solutions we adopted to overcome these problems.

4.3 APPROACH OVERVIEW

In this section we explain our approach to automatically extract a valid memory forensics profile from a memory dump. Our system can be conceptually divided in three independent phases, as illustrated in Figure 4.2. In the first phase, we find the location of all symbols in the memory dump and we identify the version of the running kernel. During the second phase we use a compiler plugin to analyze the source code of the identified version and emit a set of models – which we call *access chains* – that describe the way the code operates over a selected set of kernel objects. It is important to note that we only need access to the public source code but not to the exact configuration (kernel options, compiler settings, randomization seed, etc.) that was used to build the kernel captured in the memory dump. The chains extracted in this phase

are finally fed into the third component, the *exploration engine*, which applies them to the actual kernel binary code extracted from the memory dump. The final output of our tool is a working memory forensics profile, which can be used with Volatility to extract evidences from a memory dump.

For example, during the second phase our system would discover that the field `vm_file` of the structure `vm_area_struct` is used in the function `shm_close` *and* that the variable at the base of the access chain is the first parameter of the function.

Then, during the third phase our tool locates the aforementioned function by using the symbols extracted in Phase I, and tracks the offsets of every memory access that depends from the first parameter. This process produces the position (or a set of candidate positions which are then compared and intersected with the same information extracted from other functions) for the `vm_file` field.

4.4 PHASE I: KERNEL IDENTIFICATION AND SYMBOLS RECOVERY

The goal of the first phase is to recover two key pieces of information: the version of the kernel and the location of its symbols (functions and global variables).

Locating Kernel Functions

As we already explained in Section 3.1, existing memory forensic approaches require to know the location of certain global symbols to bootstrap their analysis. On top of that, our approach also requires the location of all kernel functions, which will serve as basis for our analysis to recover the layout of all data structures.

This recovery is greatly complicated by two different factors. First of all, unlike other memory forensics tools, we cannot rely on the knowledge of the `System.map` file, which is instead always part of a memory forensics profile. Moreover, we want our approach to be resilient

against Kernel Address Space Layout Randomization (KALSR) - which is nowadays enabled by default by several Linux distributions. To date, the forensics community already proposed several systems to recover kernel symbols from a memory dump, for example `ksfinder` [Gra16], `volatility-android` [Svi16], and the solution presented by Zhang et al. [ZMW16]. These approaches leverage the fact that some symbols of the kernel are exported using the `EXPORT_SYMBOL` macro to allow kernel modules to transparently access kernel objects and functions. Whenever a symbol is exported with this macro, the kernel initializes and inserts in the `___ksymtab` section a `kernel_symbol` structure that contains two fields: one with the virtual address of the exported symbol and the other one pointing to a string representing the symbol name– which in turn is placed in the `__ksymtab_strings` section. To locate a given symbol, all previous approaches find the physical address of the string representing the symbol by scanning the memory dump and then assume they can translate this physical address to a virtual one by adding a constant, based on the virtual base address of the kernel. With this information they are then able to scan the memory dump and match the corresponding `kernel_symbol` object.

The first problem with this solution is that exported symbols constitute only a tiny subset of all the kernel symbols. For this reason, Zhang et al. [ZMW16] introduced a way to recover the `kallsyms` - usually accessible from userspace from a file under `/proc`. However, since there are tens of thousands of symbols, in order to save space they are stored in a compressed form using a table lookup algorithm. As a result, they are much harder to located in a memory dump, and several kernel global variables are needed to decode their names. To overcome this problem, Zhang suggested to find the location of these variables from the disassembly of the function `update_iter` - which can be found by carving the corresponding `kernel_symbol`. Once these variables are found, by manually re-implementing the decoding algorithm the authors were finally able to reconstruct the `kallsyms`. Unfortunately, this approach requires a considerable manual effort and the authors did not discuss an automated way to retrieve the address of the global variables. The second, much more severe, limitations of all existing solutions is that they fail on

modern X86_64 platforms with KASLR, where both the virtual and the physical base addresses are randomized.

For these reasons, we designed a novel and generic way to automatically extract the addresses of all kernel functions and global variables. Our approach extends the ideas presented so far, but it relies on automatically finding and *executing* the kallsyms_on_each_symbol function. This function is present in the kernel tree since more than 10 years, it is exported with the EXPORT_SYMBOL macro and it is responsible to handle the symbol decoding process, making it a perfect match for our purpose. Our technique starts by carving a number of candidate ksymtab tables based on few constraints (e.g., the structure needs to include include two side-by-side valid kernel addresses, value and name, greater than 0xffffffff80000000 and at least 500 contiguous kernel_symbol objects). Since it is an exported function, we also know that the symbol representing kallsyms_on_each_symbol must be contained in one of these candidates. To find the correct one, we leverage the fact that even when KASLR is enabled, the randomization happens at the page granularity and hence offsets inside a page are left unaltered. So we scan again the memory and record every physical address matching the string kallsyms_on_each_symbol. Given the previous fact, we select the kernel_symbols that have a name pointer with the same page offset of one of the matched strings. To translate the value field from the virtual to the physical address space we leverage the fact that the kernel is always contiguously mapped in the both address spaces. Therefore, by adding the physical address of the string to the difference between the value and the name virtual addresses, we can find the physical address of a candidate kallsyms_on_each_symbol function.

Our system can now extract the function code from the memory dump and execute its code by using the Unicorn emulator [QV15]. One of the function's parameters is a callback function that is called for each decoded symbol, so we can use this to retrieve the name and the address of each symbol as it is processed.

For our experiments, we require that the kernel was compiled with the KALLSYMS_ALL, which includes in the kallsyms all the defined global

variables. While for example the Debian kernel ships with configuration, we acknowledge that this might not be always the case. For the sake of our experiments we decided to always turn on this configuration and we defer further discussions on how it might be possible to overcome this limitation to Section 4.8.

Finally, we plan to release this technique as a standalone tool or to embedded it in current memory forensics tools to effectively determine the kernel layout randomization shift.

Kernel Version Identification

Multiple techniques exist to identify the version of a kernel contained in a memory dump. The straightforward approach consists in grepping for strings that match the format of a Linux kernel banner. However, even thought the kernel is generally loaded in the first few megabytes of the physical address space and therefore the correct version should be in the first few matches, this technique can potentially result in several false positives. Because of this, we resort to a more precise identification by extracting the global variable `init_uts_ns` and the corresponding textual representation contained in the variable `linux_banner`. The location of these variables is retrieved together with all other symbols as described in the previous section. Another orthogonal approach to retrieve this information was presented by Roussev et al. [RAS14] and is based on matching fuzzy hash signatures generated from the disk image of the kernel.

4.5 PHASE II: CODE ANALYSIS

At the end of the first phase we identified the version of the running kernel, which we can use to download its corresponding source code. In this second phase we automatically analyze the code to extract three pieces of information: the type definitions, the pre-processor directives, and a list of access chain models.

```
1    int free_next(struct task *task){
2      struct task *t = task->next;
3      int gid = t->cred.gid;
4      if (strcmp(t->name, "init")){
5        free(t)
6        return gid;
7      }
8      return -1;
9    }
```

Figure 4.3: The example we use to explain how the Clang plugin works

The bulk of our analysis is performed by a custom plugin for the Clang compiler, which operates on the Abstract Syntax Tree (AST) of the Linux kernel. While the analysis we need to perform would be much easier and more practical if performed at a later stage of the compilation process – i.e. by working on the compiler intermediate representation – working on the AST provides the advantage of being compatible with *all* version of the Linux kernel. In fact, while recent versions of the kernel can compile with Clang and few older versions are supported through a set of manually created patches, for the vast majority of kernel versions Clang is not able to produce an intermediate representation. However, Clang is "fault tolerant" when it builds the AST and thus it is able to create one for all versions of the Linux kernel, also when it is not able to compile the sources.

To recover the aforementioned pieces of information, we compile the kernel configured with `allyesconfig` with our plugin, which is triggered every time an AST representing a function or a record is created. The choice of this particular configuration comes from the fact that, by turning on all the configuration options, it increases the coverage of our plugin over the kernel codebase. The actual analysis starts at the root node of a function and recursively visits the whole tree by using a depth-first strategy.

4.5.1 *Pre-processor directives*

The first piece of information we save from the compilation process is the position of `macro` and `ifdef` directives. To extract this information we use `pp-trace`, a standalone tool from the Clang framework that traces the preprocessor activity. For each statement `pp-trace` emits where it begins, where it ends and, in the case of macros, also their names. This information is used for several purposes. First of all, we will not extract chains included in `ifdef` statements, because their code is dependent on a specific configuration setting and thus might not be included in the kernel under investigation. Our tool also saves where the compiler directives related to structure randomization are used. In this way, by matching this information with the definition of a structure, our system knows which structures are affected by layout randomization. Finally, as we will explain in Section 4.6.1, by combining this information with the definition of kernel objects, it is possible for our tool to automatically deduce the offset of certain fields.

4.5.2 *Types Definition*

Along with the function's AST, our plugin also visits the AST representing the definition of kernel objects. When traversing this tree it saves the type of each object along with the name, the type, and the definition line of its fields. As a special case, when exploring unions, the tool marks the fields they contain accordingly.

For several reasons, the information gathered from parsing a definition of a record is very important. First of all, by looking at the definition order, our exploration system can constrain the candidate offsets for a given field. Moreover the offset of certain fields can be statically deduced (e.g., the first field in a structure is always at offset zero).

4.5.3 *Access Chains*

To model the way the code accesses kernel objects we introduce the concept of *access chain*, defined as a triple *{Location, Transitions, Source}*. In the triple, the *Location* defines where the access is performed, in terms of a file name, a function name, and a line number. The *Transitions* element is a list containing the type of the objects and the name of the fields of every data structure involved in the chain. For example, the chain describing the access at line 3 of Figure 4.3 would contain three elements:

```
struct task->next|struct task->cred|struct creds.gid
```

Finally, the third element of an access chain is its *Source*, that represents how the first variable of the chain was initialized. This information is essential to select the memory accesses belonging to a target object, while ignoring all the others. In the previous example, since the base variable is `task`, the source of the chain would be marked as the first parameter of the function `free_next`. Our system supports three different types of sources: function *parameters*, *global variables*, and values returned from a function invocation (*function returns*). The representation of the source depends on its category: parameters are expressed as numerical position in the argument list, while the other two categories are expressed respectively through the name of the global variable or the name of the function.

Local variables, which can be legitimately used as base variables for an access, are not valid sources. This is because local variables must be initialized before they can be used and their initialization must fall in one of the previous categories. As we will explain in the next section, a core aspect of the plugin is that it keeps a *map* from variables to their initialization. This enables the plugin to correctly determine the source for each access chain.

The plugin extract access chains from the kernel source code by parsing three types of nodes in the AST: assignments and declarations, object accesses, and function calls and returns.

Declarations and Assignments are used to maintain a map of all variables and the way they are initialized. For instance, when we encounter

the node representing the declaration at line 2 of Figure 4.3, the plugin first extracts the variable used in the left-hand side (LHS) of the statement. If the type of the variable is a `struct` or a `void` pointer, the plugin proceeds by analyzing the right-hand side of the statement (RHS). In case the RHS is already a valid source (parameter, global variable, function call) or an object access then we update the map with this information. On the other hand, if the RHS represents another local variable, then we lookup in the map how this other local variable was initialized and copy this information in the map entry of the LHS variable. This mechanism ensure that, at any given point inside a function, our plugins knows how a variable was initialized.

To simplify the analysis, it is important to note that our analysis only keeps track of one path, and not all possible paths where a variable can be assigned. However, to extract the offset corresponding to a given access is sufficient to find *one* path inside a function that reaches that access, rather than exploring *all* of them.

Object Accesses (as modeled by `MemberExpr` in Clang terminology) are the nodes that, for example, represent the right part of the statement at line 3 of Figure 4.3. Since in this case there are several objects chained together, the plugin keeps track of every field name and object type when traversing this sub-tree. When it reaches the base of the access, represented in this case by the variable `t`, a number of things can happen. If the base is a valid source itself (e.g., a parameter, a global variable, or a function) then the chain can be already emitted. Otherwise, if the base is a local variable then we recursively visit its initialization, appending in front of the chain the object types and the field names. This recursive process ends when a valid source is found and thus the chain can be emitted. For example, when the plugin traverses the sub-tree representing line 3, it first extracts the type of the object and the field name, i.e. `struct creds.gid` and `struct task→ creds`, and appends them to the chain. Then, since the variable `t` is a local variable, it checks in the definition map how this variable was initialized. Since `t` was initialized from an object access at line 2, it recursively traverses this access and it appends to the chain the element `struct task→ next`. At this point the process

ends because the base variable `task` is a valid source.

When traversing the objects involved in a chain, the plugin keeps track of how fields are accessed. While the C standard defines the arrow and the dot operator as the only way to access a field, we are interested also in other operators that may affect an access. The first is related to the `offset_of` extension and in particular to the macro `container_of`, which is built on top of it. This macro is extensively used in linked list and trees implementations, and it defines a sort of parent-child relationship between kernel objects. In fact, given a child structure and its offset inside the parent structures, the macro is used to retrieve a pointer to the parent object. For example, supposing that `c` is a pointer to a `struct creds`, the task containing it can be retrieved by calling `t = container_of(c, struct task, cred)`. A chain containing this macro needs to be treated carefully – not only because an offset is rather subtracted than added to the base pointer – but also because the compiler often merges a `container_of` element and the subsequent displacement in a single instruction. The other operator the plugin keeps track of is the reference operator (`&`). This is of particular importance when joining two chains, because it may transform an arrow in a dot operator. Finally, fields defined as array are generally accessed in a different way and thus need a particular technique during the exploration process. Therefore, if an element of a chain is a `container_of`, an array, or it contains a reference operator we mark it accordingly in our model.

Function Calls and Returns are the last two types of nodes explored by the plugin. This information is essential to extract accesses in functions which are inlined by the compiler. In case of a function call, we save the name of the called function and its arguments. Similarly to how accesses are represented, every argument is expressed as an access chain. The only difference is that these chains might have an empty *Transitions* element. This happens for example when one function calls another and it passes as parameter one of its own arguments or a global variable. A similar approach is applied to return statements.

4.5.4 *Non Unique Functions*

Another problem when dealing with projects in the size of the Linux kernel is that function names are not always unique. In fact, the `static` identifier is used to limit to a file the scope of a function. For example, this happens with the function `s_next` that, in kernel version 5.1, is defined in 5 different occasions. This is a problem for our system, because whenever we analyze a function we need to be sure that we are dealing with the correct "instance" of the function. Since there is no straightforward way to extract this information using Clang, we employed Joern [Yam+14]. This tool, among other things, contains a fuzzy parser for C and C++. The output of Joern after parsing the kernel sources, is a list of functions and the filename where they are defined. This information is used whenever our system extracts a function from a memory dump. In case the function has a non-unique name, we exploit the fact that functions defined in the same compilation unit ends up in the same object file and thus are also contiguous in the kernel binary. In this way, by checking the functions in the vicinity of the target one, our system is able to select the correct function.

Finally, for optimization reasons, the compiler can decide to remove a parameter from a function or even split a function in two or more parts. Fortunately, when these optimizations are applied, the compiler also adds a suffix - respectively `.isra` and `.part` - to the name of the function. In the first case we simply ignore the function, while in the second one our system is able to extract and join all the different pieces.

4.6 PHASE III: PROFILE GENERATION

It is important to point out that a profile includes the layout of only a small subset of all kernel data structures – those that are needed to complete the forensic analysis tasks supported by a given tool. For this reason, our system focuses on recovering only the information actually used by Volatility. However, manually listing the objects used by every Volatility plugin is a tedious and error prone process, and it is further

complicated by the fact that some of these objects vary depending on the kernel version. Therefore, for our tests we decided to instrument Volatility to print every field it traverses and then we recovered the full list by executing each plugin against a memory dump containing the same kernel version of the one under analysis.

As a result, the actual number of different fields and unique data structures vary among the experiments, ranging from 220 and 236.

4.6.1 *Binary Analysis*

To match the chains extracted during the source code analysis against the functions extracted from an actual memory dump we use `angr` [Sho+16] and its symbolic execution capabilities. Therefore, we decided to perform our exploration by symbolizing the source of a chain and run the function while tracking every time the symbolic variable is used as a base for a memory access. To avoid state explosion – one of the major problem of symbolic execution – we wrote a custom *exploration technique*. An exploration technique drives how the program is explored by deciding which states can advance and which should be discarded. In our case, it keeps track of every state generated by the symbolic execution engine and prunes those which have already been explored more than a certain amount of time, effectively limiting the state space. Moreover, we also instruct `angr` to check the satisfiability of the constraints belonging to a state as infrequently as possible, rather than checking them when a new state is created. For example, assuming two states are created from a branch instruction then both states will be kept, *regardless* of their satisfiability. These two expedients allow the number of state to be contained but also to entirely cover the code contained in a function.

While tracking the memory accesses is independent from the source of a chain, it dictates how the system is initialized and run. Parameters and function returns are the most straightforward sources to handle. In the first case a symbolic variable is stored in the corresponding register, while in the second - whenever the function specified in the source is called - we symbolized the `rax` register. On the other hand, global variables require

two different strategies to handle both pointers and normal variable. In both cases, whenever the address of the variable is stored in a register we symbolize the register itself. Moreover, when the variable is not a pointer, the compiler might have already pre-computed the address of the field. If this is the case, we directly extract the offset and append it to the list of results. Since the size of non-pointer variables is known from the `kallsyms` – by subtracting the address of the `kallsym` following the one representing the global variable – our system can discern cases where more than one non-pointer variables are accessed in the same function.

Field Dependencies – Our system often needs to deal with chains spanning multiple objects. For instance, let us consider again our sample chain:

```
struct task->next|struct task->cred|struct creds.gid
```

The code reaches the target `gid` by first traversing the `cred` pointer in the `task` structure, thus defining a *dependency* among the two fields. In other words, we first need to recover the offset of `cred` and then use that as input to extract the second half of the chain.

In this case we create multiple symbolic variables and appropriately store them when a memory access belonging to an element is detected. However, since the final assignment of a field offset is obtained by a global algorithm by majority voting, it is possible that a chain cannot be fully analyzed in one pass, but instead requires a recursive approach to first identify all its dependencies.

Nested Structures – A particular type of dependency occurs when the target field is accessed through a nested structure. In C, this may appear for example in the form of `struct_a.struct_b.target`. In this case, the compiler may split the access in two parts, by first loading the base address of `struct_b` (for instance located at `0x20` bytes from the beginning of `struct_a`) and then adding the offset of the field (e.g., `0x16` bytes into `struct_b`). However, this is often optimized by computing the total offset from the base structure at compile time, resulting in a single instruction like `lea rax, [rsi+0x36]`.

This requires our tool to keep track of this displacement, as `0x36`

is not the correct offset of `target`, and to obtain the right value we need to remove the offset of `struct_b`, which (like in the case of field dependencies) needs to be computed separately.

4.6.2 *Dealing with Inlined Functions*

Since the kernel is always compiled with the optimizations turned on, the compiler is quite aggressive when it comes to function inlining. For example, compiling the Linux kernel 5.1 with the default configuration results in the inlining of more than 200 thousands call sites. For this reason, being able to cope with function inlining dramatically increase the number of chains our exploration system can test.

When we analyze a memory dump and discover that a given function call has been inlined, we trigger a dedicated routine in charge of merging and inheriting its chains. Our process starts by labeling every chain of the target function as *forward* or *backward*. Forward chains are those that starts from a parameter, while backward ones are those that terminates in return statements. For example, in the following snippet:

```
1  inline struct task* foo(task *t, char *n){
2    t->name = n;
3    if(t->cred) {
4      return t->next;
5    }
6    else {
7      global_task->next = 0;
8    ...
```

the chain at line 2 is a forward chain, while the one at line 4 is both a forward and backward chain. Our algorithm is divided in two independent parts: in the first one chains are joined, while in the second one they are inherited.

The first one starts by iterating over every pair of caller and callee. If the callee is *not* inlined, and thus is present in the list of functions extracted from the memory dump, then no action is required. Otherwise, each argument - which is also represented with a chain - is joined with every forward chain of the callee that starts from the same position.

Joining is not a commutative operation: the source and the location of the argument chain are left untouched, while the list of objects of the callee chain are appended to the one of the argument chain. A similar treatment is reserved for backwards chain, but this time in the opposite direction. Every chain of the caller that has source equal to an inlined function, is joined with the backward chains of this function. Since the inlining depth can be greater than 1, i.e. functions called from inlined functions can be inlined as well, we repeat this process in a loop to propagate the presence of freshly joined chains, until any new chain is generated.

The second part of the process deals with inheriting from inlined functions all the chains which are not forward or backwards one, for example those who access a global object. In this case the chain is left unaltered and only added to the set of chains of the caller. In our example, as result of this process, a function that calls `foo` will have as well the chain representing the line 7. Similarly to the previous process, we also propagate inherited chains by repeating this process in a loop.

Once this two steps are finalized, our systems passes over the resulting chains to *clean* and *adjust* them. The cleaning process is needed because a target can be present in multiple same-source chains of a function. For this reason, given a target, we delete the chains which are a superset of others are deleted, thus ensuring that the target is tested only once. On the other hand, the *adjustment* deals with chains containing the reference operator or `container_of`. In the first case, we translate the arrow following a reference in a dot, but only if the chain is not used as parameter for a function. Given the following example:

```
1  struct creds *c = &t->cred;
2  set_gid(c, 0);
3  ...
4  set_gid(struct creds *c, int g){
5    c->gid = g;
6  }
```

if `set_gid` is inlined, then the compiler will most likely merge the accesses to fields `cred` and `gid` in a single one. As we explained in section 4.6.1, this chain can be explored only if the offset of either `cred` or `gid` is known. On the other hand, if the function is not inlined, no action is required

and the chain containing `cred` can be safely explored.

The adjustment of `container_of` deals with a similar problem. In the following example:

```
1  t = container_of(c, struct task, cred);
2  t->next = NULL;
```

the compiler may effectively subtract from `c` the offset of `cred` and then add the offset of `next`, or merge the previous two operations and add to `c` the distance between `cred` and `next`. In this case, to represent these two possibilities, we duplicate the chain, explore both of them and merge their results.

4.6.3 *Object Layout Inference*

At the end of the binary exploration phase, each target (i.e., each field whose position we need to locate) has its own list of candidate offsets. Since the lists associated to different fields can overlap, it is now a global optimization problem to find the set of offsets that maximizes the number of recovered fields. For instance, if three fields of the same data structure can be located respectively at offsets {72, 74}, {40, 72}, and {40} (according to our chain-matching algorithm) then we can exclude that the second field can be located at offset 40, and in turn this rules out the possibility of the first to be at offset 72.

We solve this problem by creating a `z3` model [DB08] where all the fields and respective candidates are added in the form of *soft* constrains. Moreover, in case no structure randomization is in place, we also add *hard* constraints based on the position of a field, because the order of the fields in the source code definition must be respected in the offsets layout, and because the first field in a structure always has offset zero. Special care is given to unions, since in this case we assume the fields they contain have the same offset. A problem with this approach is that if the candidates of a field are wrong and *contradict* the position constraints, then the model become unsatisfiable. To overcome this limitation, when we run into an unsatisfiable model, we explore the solution space by recursively removing a *soft* unsatisfiable constraints.

Version	Release Date	Configuration	Used Fields	Extracted Fields
4.19.37	04/2019	`Debian`	230	205 (89%)
4.19.37	04/2019	`Debian + RANDSTRUCT`	230	172 (74%)
4.4.71	06/2017	`OpenWrt`	231	198 (86%)
3.18.94	05/2018	`Goldfish (Android)`	236	204 (86%)
2.6.38	03/2011	`Ubuntu`	220	191 (87%)

Table 4.1: The Linux kernels we use in our experiments.

Finally, the *knowledge* gained from the previous modeling process is added to the system. This new piece of information will most likely satisfy the *dependency* or the *displacement* of other chains that were previously not testable. Hence, we go back and forth between the binary analysis component that resolve the chains and the layout inference component that solve the extracted candidates and constraints until no other chain is available.

4.7 EXPERIMENTS

To test our approach we collected a number of memory dumps from systems running different Linux kernels. The list of kernels (summarized in Table 4.1) was chosen to reflect different major versions (including 2.6, 3.1, 4.4, and 4.19) and different configurations. In particular, the first two experiments were conducted with the latest version of the kernel shipped by Debian, with the only difference that the second was recompiled with structure layout randomization. The last three experiments aimed instead to test our framework against less common memory forensics scenarios, when the traditional approach to create a profile would be difficult to apply. For one test we retrieved the kernel compiled from OpenWrt, a project that targets network devices; in another we recreated a scenario involving a memory dump of an Android device, and for our last test we chose an ancient version of the Linux kernel that does not support Clang.

To run our experiments we downloaded the kernel sources and configurations from the respective repositories. We then compiled two versions

of each kernel, one for testing and one to perform our source-code analysis. We configured the testing version with the configuration shipped with the distribution, and compiled it with a supported version of `gcc`. We obtained instead the second version by configuring the kernel with `allyesconfig` and by compiling the sources with our Clang compiler plugin. We then proceeded by installing the first version in a `QEMU` virtual machine, booted the machine and acquired an atomic memory dump using the QEMU console. Moreover, we also used the first version to manually create a *ground truth* Volatility profile. We were able to create this profile for each experiment except for the one using `RANDSTRUCT`. While we empirically checked that the code was correctly affected by this option and that the randomization seed was present in the kernel tree – the debugging information did not reflect the change in the structures layout. We later discovered this to be a known issue already discussed by several researchers in online forums [Bug18; - M18]. It is still unclear if the problem is due to a bug in `gcc` or in the randomization plugin, but in any case the erroneous information prevents Volatility from generating a profile even when the randomization seed is available. For this reason, for this test we decided to reuse the ground truth information from the `Debian` experiment, but this time changing the `hard` constrains associated to the randomized structures in the `z3` model during our profile extraction. While in a non-randomized setting our system could reason about the order of fields inside a struct definition, now it could only assert that two fields needs to have a different offset – or the same one if they are part of a union.

Building the profiles using our automated approach took approximately eight hours in each experiment. The first phase was the fastest and the only one that depends on the size of the memory dump. Our prototype required around three minutes to analyze `4GB` of memory and retrieve all kernel symbols. The static analysis performed in Phase two took three hours on a eight-core machine. In this phase, most of the time is spent compiling the kernel configured with `allyesconfig`. Finally, the exploration of kernel functions using `angr` and the generation of the final profile is the most time-consuming phase of our experiments and took in

average five hours on a cluster of 64 cores.

4.7.1 *Results*

The fourth column of Table 4.1 shows how many unique fields are used by Volatility for the given image. The value range from 220 to 236 but, quite surprisingly, the intersection of these fields counts almost 180 elements. This means that, even if new features frequently land in the kernel tree, a large fraction of fields used by memory forensics is not affected by the kernel development. These fields are mostly related with process management (e.g., `task_struct`), process memory (e.g., `mm_struct` and `vm_area_struct`), and filesystem information (e.g., `dentry` and `file_operations`). The last column of Table 4.1 shows instead how many fields our framework was able to correctly extract in the generated profile. The recovery rate ranged from 74 to 89%, but this value alone does not tell us much about how many Volatility plugins are working with the extracted profile. In fact, in most of the cases it is enough that one field was wrongly extracted to undermine the result of an entire plugin.

To answer this question, Table 4.2 breaks down, for each plugin, the number of fields that were correctly located by our system and the number of fields for which we extracted a wrong offset. Unfortunately, it is not sufficient to compare the list of fields accessed by a plugin to tell which plugin is correctly supported by our profile. For example, our instrumented version of Volatility reports that the plugin `linux_pstree` accesses the field `gid` of `struct cred` but this information is never used in the analysis. Therefore, we decided to compare two runs of Volatility against the same memory dump: one by using the profile extracted by our framework and the other by using the one we manually created. The result of this comparison is shown in 'Working' columns in Table 4.2. Each cell represents whether our profile contains all the necessary information for a given plugin (●) or not (○). In addition, in certain cases it is possible that, even if one field was not correctly extracted, the plugin is still able to function with reduced functionality (◑). For example, this happens in

	Debian 4.19			RANDSTRUCT			Openwrt			Android			Ubuntu 2.6		
	Working	Correct	Wrong	Working	Correct	Wrong	Working	Correct	Wrong	Working	Correct	Wrong	Working	Correct	Wrong
linux_arp	◑	10	2	○	6	6	◑	10	2	◑	11	1	●	12	0
linux_banner	●	0	0	●	0	0	●	0	0	●	0	0	●	0	0
linux_check_afinfo	—	5	1	—	5	1	●	37	3	●	40	2	○	34	5
linux_check_creds	●	9	0	●	9	0	●	9	0	●	9	0	●	9	0
linux_check_fop	○	77	4	○	65	16	○	75	4	○	76	2	○	67	3
linux_check_modules	○	17	1	○	14	4	○	15	2	○	17	0	○	15	2
linux_check_syscall	●	36	1	●	32	5	◑	31	5	●	33	3	●	32	3
linux_check_tty	○	11	3	○	8	6	○	9	4	○	9	4	○	11	2
linux_cpuinfo	○	0	2	○	0	2	○	0	2	○	0	2	●	2	0
linux_dump_map	●	10	0	○	6	4	●	10	0	●	10	0	●	9	1
linux_dynamic_env	●	29	0	●	23	6	●	6	0	●	29	0	●	27	1
linux_elfs	●	26	0	○	20	6	●	25	0	●	26	0	●	23	4
linux_enumerate_files	●	24	0	○	21	3	○	23	1	●	24	0	○	22	1
linux_find_file -L	●	24	0	○	21	3	○	23	1	●	24	0	○	22	1
linux_getcwd	●	16	0	●	15	1	●	10	0	●	16	0	●	16	0
linux_hidden_modules	○	7	1	○	6	2	○	3	4	○	4	3	○	5	2
linux_ifconfig	○	10	2	○	10	2	○	11	1	○	10	2	○	9	3
linux_info_regs	—	10	0	—	10	0	●	11	0	◑	10	1	●	11	0
linux_iomem	●	5	0	●	5	0	●	5	0	●	5	0	●	5	0
linux_keyboard	●	1	0	●	1	0	●	1	0	●	1	0	●	1	0
linux_ldrmodules	●	32	0	○	25	7	○	30	1	●	32	0	●	29	4
linux_library_list	●	11	0	○	7	4	●	10	0	●	11	0	●	10	4
linux_librarydump	●	11	0	○	7	4	●	11	0	●	11	0	●	10	1
linux_list_raw	○	33	2	○	31	4	○	4	2	○	33	2	○	30	4
linux_lsmod	○	5	1	○	5	1	○	3	2	◑	3	2	○	4	1
linux_lsof	●	24	0	◑	22	2	●	24	0	●	24	0	●	23	0
linux_malfind	●	17	0	○	16	1	○	16	1	○	17	0	○	16	1
linux_memmap	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0
linux_moddump	○	4	7	○	4	7	○	4	7	○	5	6	○	4	6
linux_mount	●	20	0	○	18	2	●	20	0	●	20	0	●	19	0
linux_netscan	●	16	1	●	16	1	●	15	2	●	15	2	○	14	3
linux_netstat	○	28	2	○	27	3	○	28	2	○	28	2	○	27	3
linux_pidhashtable	○	22	2	○	21	3	○	22	5	○	22	5	○	19	2
linux_plthook	●	26	0	○	20	6	●	24	0	●	26	0	●	22	4
linux_plthook -a	●	26	0	○	20	6	●	24	0	●	26	0	●	22	4
linux_proc_maps	●	37	0	○	30	7	◑	36	1	●	37	0	◑	34	1
linux_proc_maps_rb	○	38	1	○	33	6	○	37	2	○	38	1	○	35	2
linux_procdump	○	6	1	○	7	0	○	6	1	○	6	1	○	6	1
linux_psaux	●	13	0	●	12	1	●	13	0	○	12	1	●	11	0
linux_psenv	●	8	0	●	8	0	●	8	0	●	8	0	●	8	0
linux_pslist	◑	17	1	◑	16	2	●	19	0	◑	16	3	◑	12	1
linux_psscan	●	12	1	●	11	2	●	13	0	●	13	0	●	12	1
linux_pstree	●	13	0	●	11	2	○	11	2	○	11	2	●	11	0
linux_threads	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0
linux_tmpfs -L	●	20	0	○	18	2	●	20	0	●	20	0	●	19	0
linux_truecrypt	●	3	0	●	3	0	●	3	0	●	3	0	●	3	0
Total Working Plugins	**29**			**15**			**25**			**27**			**27**		

Table 4.2: Symbol ● denotes plugin is working, ○ not working, ◑ partially working, and — not supported by volatility

3 out of 5 cases for the `linux_pslist` process where all the information except the creation time are listed correctly by using our profile. Finally, cells containing the dash sing (-) denotes that the corresponding plugin was not supported on the kernel under analysis.

The only exception to this approach are those plugins that do not produce any output in our tests. For example, the `linux_malfind` plugin searches for traces of malware infection but if the machine is not infected – like in our experiments – then the plugin does not produce any output. Similarly the plugin `linux_hidden_modules` searches for kernel modules that were un-linked from the modules list. Therefore, for these cases we resorted to check that the offsets of all fields accessed by the plugins were correctly recovered to determine whether the plugin was supported or not by our profile.

Overall, on the non-randomized memory dump, between 57% (for OpenWRT) and 64% (for Debian) of the plugins worked correctly with our profile. In particular, the profile automatically created by our framework was able to support many plugins which are fundamental for a forensics analysis. This include the support to extract the list of running process and many related information such as their memory mappings, credentials, opened files, and environment variables. Moreover, our profile can be used to successfully recover the content of `tmpfs` and to retrieve the list of open network connections in 4 out of 5 memory dumps.

In other cases, our system was not able to recover the right offsets for the required fields. For instance, the fields related to the list of modules and to the network interfaces were much harder to extract correctly. However, even if we report the plugin as not supported in the results of Table 4.2, in practice an analyst could often overcome this limitation by testing different profiles. For instance, for the field `in_ifaddr->ifa_label`, used by `linux_ifconfig`, our system extracted two possible offsets, one of which was the correct one. In other words, our technique could be used to generate two profiles and simply ask the analyst to try both during the analysis. Overall, the number of models that contains two or three offsets – one of which is the correct one – accounts for the 41% of missing fields.

Finally, the experiment on the randomized kernel shows that the *hard*

constraints play an important role in our system. More than 140 or the 230 fields used by Volatility are contained in structures affected by layout randomization, and currently our system is able to correctly extract the offset of 60% of them. However, our system failed to distinguish the offsets of the `vm_start` and `vm_end` fields of `vm_area_struct`, thus impacting the result of a large number of plugins that reason on the mappings of a process and its loaded libraries. While we could not count them as supported by our fully-automated approach, in reality for an analyst it is trivial to distinguish among the two fields (as one is always larger than the second) – again showing that even when the profile could not be directly used to perform the analysis, it can still serve as an input for simple manual adjustments and refinements.

4.7.2 *Chain Extractions*

Table 4.3 shows detailed statistics about our analysis. Because of space constraints we could not include all 230 fields and we decided therefore to limit the table to the fields belonging to `mm_struct` and `vm_area_struct`. For each field, the table reports the number of chains extracted in Phase two, the number of chain explored in Phase three, the number of chain that contained at least one dependency or displacement not satisfied, and finally the number of offsets generated by our system (a ✓ sign means that the tool identified the right offset, while a number means that the model was not only containing the correct offsets, but also other possible candidates). There are several interesting information that can be deducted from this table. First of all, both the `mmap` and the `vm_start` fields were never explored, because they are the first field of the respective structures and thus their offset (zero) was automatically deducted. Moreover, it shows the first two iterations of our recursive approach. For example, the model of `start_brk` contained four candidates at the end of the first step because some chains were not analyzed as they depended on the offset of other fields that were still unknown. However, at the second iteration our system was able to analyze seven more chains, and

Target	Step 1					Step 2				
	Total	Expl.	Dep.	Disp.	Model	Total	Expl.	Dep.	Disp.	Model
mm_struct→ mmap	—	—	—	—	✓	—	—	—	—	✓
mm_struct→ arg_end	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ arg_start	6	3	3	0	✓	3	3	0	0	✓
mm_struct→ brk	9	2	7	0	✓	7	7	0	0	✓
mm_struct→ context	34	2	19	13	✓	32	0	0	32	✓
mm_struct→ env_end	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ env_start	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ mm_rb	13	7	6	0	✓	6	6	0	0	✓
mm_struct→ owner	8	3	5	0	✓	5	5	0	0	✓
mm_struct→ pgd	77	64	13	0	✓	13	11	2	0	✓
mm_struct→ start_brk	8	1	7	0	4	7	7	0	0	✓
mm_struct→ start_code	5	4	1	0	2	1	1	0	0	2
mm_struct→ start_stack	7	1	6	0	✓	6	6	0	0	✓
vm_area_struct→ vm_start	—	—	—	—	✓	—	—	—	—	✓
vm_area_struct→ vm_end	158	127	31	0	✓	31	22	9	0	✓
vm_area_struct→ vm_next	57	48	9	0	✓	9	8	1	0	✓
vm_area_struct→ vm_mm	135	126	9	0	✓	9	5	4	0	✓
vm_area_struct→ vm_flags	198	180	18	0	✓	18	15	3	0	✓
vm_area_struct→ vm_pgoff	100	92	8	0	✓	8	6	2	0	✓
vm_area_struct→ vm_file	130	124	6	0	✓	6	5	1	0	✓

Table 4.3: An excerpt of the fields used by Volatility and some statistics associated to their exploration

that additional information was sufficient to narrow down the choice to a single, correct, offset.

4.8 FUTURE WORK

We believe there are several ways to improve the accuracy our framework. First of all, in the phase one, we assume that the kernel was compiled with the option `KALLSYMS_ALL`, so every global variable is included in the `kallsyms`. This configuration was turned on by default in 3 out of 5 experiments, but we acknowledge that it might not always be the case. A way to solve this problem can be to use the information produced by the compiler plugin to extract from the kernel binary itself the address of the missing global variables.

Moreover, a major improvement to our framework can come from saving more details about an access chain. For example, we believe it is possible to statically determine in our compiler plugin if a particular field is only read or also written, thus filtering memory accesses belonging to one type or the other. Finally, also exploring *what* is written inside a particular field - for example a constant or a parameter - or *how* the content read from the field is used will reduce even more the number of candidates extracted from a function.

5 BACK TO THE WHITEBOARD: A PRINCIPLED APPROACH FOR THE ASSESSMENT AND DESIGN OF MEMORY FORENSIC TECHNIQUES

5.1 INTRODUCTION

Modern operating systems are very complex pieces of software whose memory often contains millions or tens of millions of individual objects at any moment in time. Even worse, both the fields and the layout of these objects can change when the kernel is updated or recompiled, and the connections among them evolves very rapidly – with a considerable amount of links and pointers that change every few milliseconds.

Currently memory forensics techniques rely on a large number of rules and heuristics that describe how to navigate through this giant graph of kernel data structures to locate and extract information relevant to an investigation. For example an analyst can use rules — commonly known as *plugins* in the field terminology — to retrieve the list of processes running at acquisition time (including their name, starting time, process ID, and other related information) or the list of open sockets. The result of the analysis depends on the number and accuracy of these rules. However, the field today is still in its infancy and each individual technique is manually written by researchers and practitioners. As a result, it is often unclear why a particular exploration strategy has been chosen, except for the fact that some developers found it reasonable based on their experience. Even worse, how accurate a given heuristic is and how we can compare it with other candidates to decide which strategy is more suitable for a given investigation remains an open question.

In fact, we still do not even know how to properly characterize the accuracy of a technique, as its quality depends on the metric we use to evaluate it, which in turn depends on the goal of the analyst. For instance, in an adversarial environment in which the analyst is investigating a sophisticated attack, a good heuristic would be one that is difficult to

evade for an attacker. In a different investigation in which there is no risk of tampered kernel data, a heuristic that only traverses closely-related (in physical memory) data structures may be preferable as pages acquired far apart may otherwise contain inconsistent information if the dump was not acquired atomically.

Contribution: The goal of this paper is to introduce a more principled way to approach the problem of memory analysis and forensics. Our plan is articulated around three main points. The first intuition is that heuristics used to extract information from memory dumps should be automatically generated by computers and not handpicked by humans. As we will show in our experiments, the graph of kernel objects is tightly connected and there are tens of millions of different ways to reach a given structure by starting from a global symbol. The second point is the fact that it is very important to be able to quantitatively measure the properties of each heuristic, so that different options can be compared against one another and an analyst can decide which technique is more appropriate for her investigation. Finally, the analyst should be able to obtain some form of guarantee about the results, to ensure that once a given quality metric has been chosen, a certain technique is the *optimal* solution to navigate the intricacies of runtime OS data structures.

As a step towards these goals, we constructed a complete graph of the internal data structures used at runtime by the Linux kernel. In our graph, nodes represent kernel objects and edges a pointer from one object to another. We chose Linux as the availability of its source code simplifies the creation of our model. However, a similar graph was also extracted in the past by Microsoft for the Windows kernel [Car+09] and could therefore be reused for our purpose. The resulting map of the memory is a giant network (containing over a million nodes) with a very dynamic topology that is constantly reshaped as new data structures get allocated and deallocated.

Memory forensics tools adopts rules to navigate through the data structures present in a memory dump, and these rules can therefore be represented as paths in our kernel graph. Nodes and edges can then be decorated with additional pieces of information that capture different

properties an analyst can find important in an analysis routine. In our study we model this phase by introducing and discussing five different metrics: *Atomicity*, *Stability*, *Generality*, *Reliability*, and *Consistency*. We used these metrics to compute a score associated to each path, and therefore to existing memory analysis techniques, as well as to compute the optimal solution according to a chosen set of criteria. We then discuss the intricacies of identifying such optimal paths by performing experiments with 85 different kernel versions and 25 individual memory snapshots acquired at regular time intervals.

Building a map of the kernel memory is a very tedious and time-consuming process. However, we believe this map can have many interesting uses in computer security beyond memory forensics – including virtual machine introspection (VMI), kernel hardening, and rootkit detection. Since both the code and the results of the previous attempts to build this graph [Lin+11; Car+09; Ibr+13] are not publicly available, we decided to release all our data – hoping it will help other researchers to considerably reduce the time required to investigate and validate techniques that require information about the content of a running kernel.

5.2 MOTIVATION

Being quite in its infancy, memory forensics has still many open problems, which have been recently summarized by Case and Richard [CR17]. The authors divided them in two categories, depending on whether they are related to the *acquisition* or the *analysis* of a memory dump. More precisely, the first category contains all the practical issues of acquiring memory from a device under investigation while the second one deals with the capabilities of memory forensics, such as malware detection and evidence extraction.

One of the main issues belonging the first category is *page smearing*, which is a consequence of the fact that while the acquisition is performed the underlying system is not frozen and thus the dump may contain inconsistent information [GF16]. While the term was coined in 2004 [Car15], its actual implications are still unclear to the community. For instance, a

recent study from Le Berre [Le 18] pointed out that in real investigations more than the 10% of memory dumps suffer from this problem and thus can not be properly analyzed with existing tools. Our work can help mitigating this issue by assessing how existing techniques are affected by non-atomic acquisitions, and help design new heuristics which are more robust against the presence of inconsistent information.

The second category focuses instead on challenges related to memory analysis. For example, as of today, forensics practitioners lack the necessary tooling for extracting a number of interesting information such as Powershell activity and evidences related to Office applications and private browsing sessions, or to analyze sophisticated userland malware. Finally, a vast range of technologies did not receive any forensics coverage: Apple iOS, Chromebooks, and IoT devices are still out of scope when it comes to memory forensics analysis.

While these issues are very different from one another, most of them share the same underlying assumption: kernel objects must be located, traversed and interpreted by a set of rules. Our approach enables forensics practitioners and researchers to evaluate, under different constraints, the quality of these rules and provide them with a framework to compare and discover new sets of rules.

5.3 APPROACH

In this section we describe the four-step approach we propose to precisely measure and improve the quality of existing memory forensics techniques. The first step consists of building a precise representation of all data structures that exists in a running kernel and of the way these structures are connected to one another. The challenges and the process we followed to build this kernel graph are described in details in Section 5.4. In the second phase we map existing forensic analysis techniques into our model, by representing their algorithms as paths through the kernel graph. We then color the graph according to different properties that are relevant for a forensic investigation, and we employ graph-based algorithms to assess the characteristics of the previously-identified paths

① `[init_task].tasks.prev` ↺ → `task_struct.mm` →
`mm_struct.mmap` → `vm_area_struct.vm_next` ↺ →
`vm_area_struct`

② `[root_cpuacct].css.cgroup` →
`cgroup_root.cgrp.e_csets[2].next` → `css_set.tasks.next`
→ `task_struct.mm` → `mm_struct.mmap` →
`vm_area_struct.vm_next` ↺ → `vm_area_struct`

Figure 5.1: Two different paths that reach the same `vm_area_struct` object.

and find new ones that may exhibit better properties. Finally, in the fourth and final phase of our methodology we translate our findings back to the memory forensic space by generating improved analysis plugins, thus increasing the number and quality of the rules that are used today to analyze memory dumps.

5.3.1 *Memory Forensics as a Graph Exploration Problem*

The goal of memory forensics is to bridge the semantic gap between the raw bytes that constitute a physical memory's snapshot and the high-level abstractions provided by modern operating systems. This task requires the forensic tool to be able to correctly translate virtual to physical memory addresses, as well as to identify the data structures that contain the required information (e.g., the name of the files opened by a given process). The latter is typically achieved in two phases. First, the system locates a known object – either because it resides at a fixed or predictable location, by using symbols information generated by the compiler when the kernel was built, or by carving a particular data structure based on a set of known properties and invariants. Starting from this entry point, the analysis then traverses different memory regions, moving from one data structure to the next by following pointers, until it reaches the required piece of information. For example, we assume the analyst found a sus-

picious process and she wants to extract its executable code for further analysis. On Linux, this analysis starts from extracting the position of the global kernel variable `init_task` of type `task_struct`. This is one of the most important kernel object in terms of Linux memory forensics since every kernel thread and user space process has its own and it serves as a hub to reach several other relevant pieces of information. After locating `init_task`, the processes list is walked until the `task_struct` belonging to the suspicious process is found. From here, the `mm_struct` is reached by dereferencing the `mm` field. Finally, the list of `vm_area_struct`, each of which defines a virtual memory area, is retrieved — first by following the `mmap` pointer, then by using the `vm_next` field. With this information, the analyst can find the executable regions of the process and can proceed to save their content to disk.

This procedure can be naturally represented as a path on a graph in which every node is a kernel object, and every link a pointer. While the final node is dictated by a given forensic task, both the first and the intermediate nodes are often the result of handcrafted routines based on the experience and expert judgment of the developers of the forensic tool.

In our graph, the previously presented analysis would correspond to the path ① in Figure 5.1. The path contains the names of the structures and fields that need to be traversed (in square brackets when they refer to global symbols in the kernel) as well as the type of transition (→: follow a pointer reference, ↻: visit multiple structures of the same type linked together). For simplicity, we report inner structures in our paths as names in the edge and not explicitly as standalone nodes. Also, note that in the example ①, since the suspicious process was freshly spawned, the shortest path in our graph traverses the process list *backwards* — contrarily to the more common *forward* walking.

On top of the previous solution, our approach shows that a stunning 2.5 million different sequences of vertices exist in the kernel graph to reach the very same target object starting from a global variable, *only* counting the paths with no more than 10 edges. For example, path ② in Figure 5.1 begins from the little-known global symbol `root_cpuacct`, passes through a number of *cgroup*-related objects, before finding the `task_struct` of the suspicious process.

The previous two "rules" are both capable of locating a given process structure in a memory dump. The first is certainly more intuitive and it may also traverse a lower number of data structures. However, this is purely a *qualitative* assessment, and it is unclear if the first solution actually has any clear advantage or whether it provides any better guarantee then the second.

5.3.2 *Path Comparison*

As we saw in the previous example, if we want to assess the quality of a given solution, we first need to define what "quality" means in our context. In other words, when two paths exist to reach the same target data structure, we need to define a metric that can tell us which one is better to follow from a forensic perspective.

A developer may favor the shortest path, as it is simpler to implement and may appear to be more robust according to the intuition that the fewer the data structures that need to be parsed, the less likely it is that something can go wrong while doing that. However, this approach raises another important issue about today's approach for memory analysis: its ad hoc nature and lack of a scientific foundation. In fact, it is not clear today how different exploration techniques can be compared and how they can be evaluated against one another in a precise and measurable way.

A first important observation is that there is not a single, absolute metric that defines the quality of a memory exploration rule. It all depends on the goal of the analyst, the conditions under which the memory snapshot was acquired, and the type of threat that is investigated. For example, in the common case in which a memory snapshot is acquired non-atomically, the analyst may prefer to adopt an approach that only traverses structure closely located in memory, thus minimizing the chances of inconsistencies. On the opposite case in which the memory was acquired atomically in a lab from a virtual machine used to investigate a possible rootkit, the analyst would certainly favor a different approach that traverses structures whose values cannot be tampered with by the attacker. In yet another

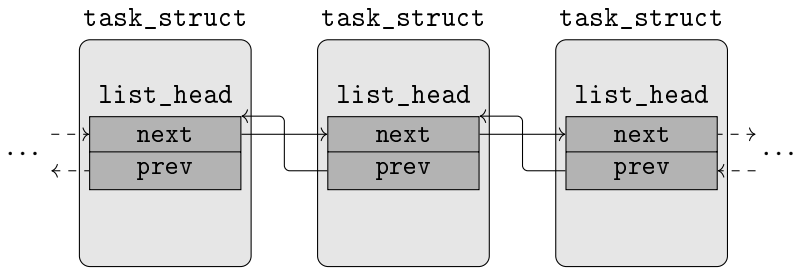


Figure 5.2: `task_structs` organized in a doubly linked list.

scenario, an investigator may try to analyze a dump for which she was not able to retrieve a correct OS profile, and therefore she might be interested in paths that traverse structures that have changed very rarely across different kernels, to maximize her probability of success.

Therefore, it is the analyst who needs to select the more appropriate fitness function to compare paths according to any combination of desired properties. And once this function has been chosen, it is possible to use it to compute the optimal path (and therefore the optimal exploration strategy) to traverse the kernel graph. In this thesis we explore different possible scenarios by proposing several metrics to enrich the graph (more details about this process are presented in Section 5.5) and then use this information to evaluate existing approaches and discuss other, non-conventional solutions that can provide better guarantees for the analyst.

5.4 GRAPH CREATION

The first step of our methodology consists in building a model of the operating system kernel, that we can later use to compare different memory forensic approaches. The model we chose for our analysis is a graph of kernel objects, in which nodes represent kernel data structures and edges represent relationships between objects (for example a pointer from one structure to another).

The core idea is simple and relies on two crucial pieces of information extracted from the kernel debugging symbols. The first one is the layout,

in terms of the exact type and offset of each field, of all the `struct` defined and used by the kernel code. The second information is instead related to the address, name, and type of global kernel variables that play the role of entry points for our graph exploration. Starting from these global pointers, our algorithm can recursively traverse other structures, each time following a pointer and casting the target memory to the appropriate type. While this process may seem straightforward at first, there are many special cases that make the construction of a kernel graph a complex procedure that requires multiple phases and several dedicated components.

In the rest of the section we discuss in more details some of these problems and the way we handled them in our study: abstract data types (and the issue with non-homogeneous circular lists), opaque pointers, and the presence of uninitialized or invalid data.

5.4.1 *Abstract Data Types*

Over the years, to maintain a reasonable quality over its code base, the Linux kernel developers have adopted several design patterns [LWN09]. In particular, the kernel exports a rich set of APIs to manipulate and create complex data structures, such as double-linked lists and trees of various types, thus relieving kernel developers from the burden of reinventing the wheel every time they need to store and organize multiple objects. For this reason, the existing APIs are not tied to a specific type of kernel object but rely instead on predefined data types that can be included in more complex `struct` objects, and in a number of macros to manipulate them.

Figure 5.2 shows one of the most common example of this pattern, in which several `task_struct` are organized in a doubly linked list using the `list_head` type. While this provides a simple and efficient way to organize data structures, it unfortunately poses a serious challenge to the automated exploration of kernel objects. In fact, if the leftmost `task_struct` in the figure was already identified by other means (for example because it was pointed to from a global variable), simply following

the `next` pointer would result in the discovery of the inner `list_head` structure, but *not* of the outer `task_struct`.

In fact, this operation is performed in the source code by using dedicated macros. In the case of the previous example, a developer would invoke:

```
container_of(var, struct task_struct, task)
```

that the compiler pre-processor translates to a snippet of code required to cast the target `list_head` variable `var` to the requested type based on the current offset inside it (as specified by the field `task`). However, in our analysis we cannot simply mimic the same behavior by subtracting the offset of the list field from `next` pointer and to cast the result to the correct type to obtain a reference to the outer object. In fact, there are many cases in which this approach would lead to wrong results and it is not sufficient to look at the field type or at its value to distinguish these problematic cases. One example is the list rooted in the field `children` of a `task_struct`. While the field points to another `task_struct`, it does so by reaching it at a different offset (in the `sibling` field). Because of this and other similar problems (explained in more details later in the paper) it is not possible to systematically apply the "subtract and cast" strategy.

For each pointer in a data structure we need to know where — in terms of object type and offset in the target structure — it points to. Other works that built a map of the Linux kernel [PH07; BGI08; XCB09] solved the problem by manually annotating the source code. While this was doable for old kernel versions (e.g., 2.4), it would take many weeks of tedious work to annotate a recent kernel – which today uses more than 6000 different data structures and more than a thousand instances of `list_heads`. Moreover, manual annotations are error prone and are tailored to one specific code base, thus requiring to be verified and modified whenever a new kernel version is released. The compiler community has also already extensively studied the points-to problem [Das00; HL07; HT01; PKH07; Ste96; WL95]. Unfortunately, the techniques they proposed are not suitable to our work as they tend to favor speed (an important factor at compile-time) over precision [Car+09] (a more important factor for our analysis). Only four previous studies automatically extracted a

type graph of a kernel [Car+09; Lin+11; Ibr+13; SPE12]. However, none of their systems is available: in one case because the authors relied on the internal source code of the Microsoft Windows operating system [Car+09], and in the other because the entire work was lost [Ibr+13].

For this reason, we decided to implement our own points-to analysis – which consists of a `clang` plugin that reasons on the Abstract Syntax Tree (AST) of each kernel compilation unit. Contrary to standard points-to analysis, our approach focuses only on the *type* information. More precisely, traditional solutions are designed to identify where each pointer points to, while in our case we only need to extract the target structure, and the offset inside that structure. The result is a *type graph* of the kernel under analysis. To extract this information we take advantage of the fact that the information we need can be inferred by analyzing the source code of the kernel that is in charge of manipulating the data structure in question. Our plugin explores the AST until it finds a call to a kernel API related to data structure management. At this point it analyzes the parameters and resolves their structure type and field name. An example of API call and respective AST is given in Figure 5.3. In the example, a call to the API `list_add` is used to append the `new` task at the beginning of the list rooted at `head->tasks`. This give us the information that the field `tasks` of `task_struct` indeed points to the very same type. Our plugin current supports `list_heads`, `hlist_heads` (used in the implementation of hash tables), and `rb_root` (used in the implementation of red-black trees).

Except for those, the most common type that is still not supported by our prototype is `radix_tree`, which however is only used 8 times in the entire kernel code base.

As we will show in Section 5.4.6, our approach is very effective and was able to resolve the type pointed by 250 global lists and by more than 1110 unique object fields in the Linux kernel 4.8, compiled with the Ubuntu 16.04 kernel configuration. Moreover, while our approach is tailored to the Linux kernel, it can be adapted to work on any other operating system, given the availability of its source code. Finally, since the parameter resolution routine does not perform complex analyses, our

```
int foo(..){
  struct task_struct *head;
  struct task_struct *new;
  ...
  list_add(new->tasks, head->tasks);
}
```

```
CallExpr 'void'
|-ImplicitCastExpr 'void (*)(struct list_head *, struct list_head *)'
| `-DeclRefExpr 'void (struct list_head *, struct list_head *)' Function
    'list_add'
|-UnaryOperator 'struct list_head *' prefix '&'
| `-MemberExpr 'struct list_head':'struct list_head' lvalue ->tasks
|   `-ImplicitCastExpr 'struct task_struct *' <LValueToRValue>
|     `-DeclRefExpr 'struct task_struct *' lvalue ParmVar 'new
`-UnaryOperator 'struct list_head *' prefix '&'
  `-MemberExpr 'struct list_head':'struct list_head' lvalue ->tasks
    `-ImplicitCastExpr 'struct task_struct *' <LValueToRValue>
    `-DeclRefExpr 'struct task_struct *' lvalue Var 'head'
```

```
[POINTS_TO] struct task_struct.tasks -> struct task_struct.tasks
```

Figure 5.3: On the top a call to `list_add`, in the center its simplified AST representation, and on the bottom the plugin output.

analysis does not introduce any significant overhead at compilation time.

Circular Lists of Non Homogeneous Elements

As we already discussed in the previous section, certain linked list can chain together object of different types. Since the code must have a way to determine to which type the target element belongs to, this pattern is only present in the form of a "root" object which is the first element of a circular list of otherwise homogeneous objects.

As a consequence, these lists can *only* be traversed starting from their root node, as traversing the loop from an intermediary objects can result into unexpectedly reaching the root node (of a different type) when dereferencing one of the `next` pointers. For example, other than the already cited **children** field of `task_struct`, also the `thread_node` field of the same structure points inside a `signal_struct` object.

To avoid this problem, our analysis classifies every `list_head` field in one of the following three categories: *root* pointer, *intermediate* pointer or *homogeneous* pointer. The first two are used to mark `list_head` fields that belong to lists that contain mixed types, while the latter describes the more common case of homogeneous list. For instance, `task_struct.children` is a root pointer, `task_struct.sibling` an intermediate pointer and `task_struct.tasks` a homogeneous one. This classification can be automatically derived from the type graph: whenever two objects of different types are involved we label the first as *root* and the second as *intermediate*, while all the other objects are labeled as *homogeneous*. During the exploration phase, depending on the type of the pointer, we adopt a different strategy:

- *homogeneous* pointers can be explored by our algorithm in any order.
- *root* pointers require instead our algorithm to immediately walk and retrieve the objects of the entire circular list.
- *intermediate* pointers are ignored since we do not know if they point to another intermediate element or to a root head. This case happens when we enter a circular list from one of its middle elements. This pointer will eventually be explored when the corresponding root node will be visited.

This classification works for every list encountered during the exploration phase, *except* for global `list_head` variables which are always marked as *root* node. In this case, during the very first part of the exploration, these lists are walked entirely and their elements appended to the worklist.

5.4.2 *Uninitialized and Invalid Data*

During our data structure exploration, there are cases that could potentially introduce false nodes to our graph. This is due to pointers that contain valid memory addresses but are not yet initialized or that were not valid at the time the snapshot was acquired. One common cause for

these errors is the fact that most of the memory management kernel APIs do not initialize to zero the allocated memory. As a result, if an object contains an array of pointers there is not way to tell if one element points to an initialized object (except if the pointer has an invalid value). Another source of false-positives comes from the *non quiescent* state in which the kernel might be when the snapshot is taken [Hof+11]. In other words, this means that the kernel could have been in the middle of updating a data structure, leaving dangling pointers in the snapshot. Finally, even if very rare, kernel bugs can contribute to the generation of similar errors.

For these reasons we implemented two sets of heuristics to check if an object is valid or not. The first *soft* rule checks that the number of valid pointers in a kernel object is greater or equal than the number of invalid ones (after removing null pointers and the pointers which normally point to userspace memory, such as the ones contained in `struct sigaction`). The second, more precise, heuristic immediately flags an object as invalid if certain conditions are not verified (such as kernel objects that contain a negative spinlock, or those with function pointers that do not point in the executable sections of the kernel). Finally, we require that, whenever present, a `list_head` has to be valid, i.e. its `next` and `prev` pointer must point to addressable memory. If these rules are not met, we consider the object invalid and discard it from our analysis.

5.4.3 *Opaque Pointers*

Opaque pointers, as represented by `void*` fields or by `long long` integers that contain at runtime the address of other objects, are traditionally one of the hardest obstacle to build a complete map of kernel objects. Luckily, this is not the case in our particular scenario. Since we are interested in using our graph to analyze and improve existing memory forensic techniques, opaque pointers play a very marginal role (if any at all) in this space. As they can point to potentially any structure, and the actual target type can change over time, traversing these pointers can be unpredictable during a post-mortem analysis. Even if none of the heuristics we encountered in our experience make use of them, we

decided to include them in our graph. After the exploration ends, in case the target of an opaque pointer was discovered by other means, we create the resulting edge, clearly marking it. In this way, we are able to detect if any of these edges are traversed during our experiments. Finally, it is important to understand that these limitations cannot lead to "wrong" results (since they cannot create erroneous paths in the graph), but nonetheless restrict the guarantees of optimality we discuss in the next sections to the constructed graph.

5.4.4 *Limitations and Manual Fixes*

Like all previous attempts to build a map of the kernel memory, two particular limitations also affect our solution: unions, and dynamically allocated arrays. Handling the latter case would require more sophisticated code analysis techniques to identify the variable number of elements contained in the arrays, which are beyond the scope of this paper. Nevertheless, we identified few cases of dynamically allocated arrays that contain information that can be relevant for memory forensics and we decided to handle them by hardcoding a custom logic. The first cases are global hash tables where often the size is not inferable from the hash table itself. For example, the `pid_hash` hash table, used by the kernel to quickly locate a process given its process id, is implemented by using a dynamically allocated array where the size is specified in another global variable (`pidhash_shift`). The second cases are instead dynamically allocated arrays pointed by a kernel object. For example, the file descriptor table associated with each process, which contains the files opened by a process (field `fd` of `struct fdtable`). Once again, this is a dynamically allocated array of `struct file` pointers, and the size can be retrieved from the field `max_fds` of the same structure.

Finally, the handling of `per_cpu` pointers was also hardcoded in our implementation. These are special pointers that, thanks to a double indirection mechanism when dereferenced, give to each processor a different copy of the same variable.

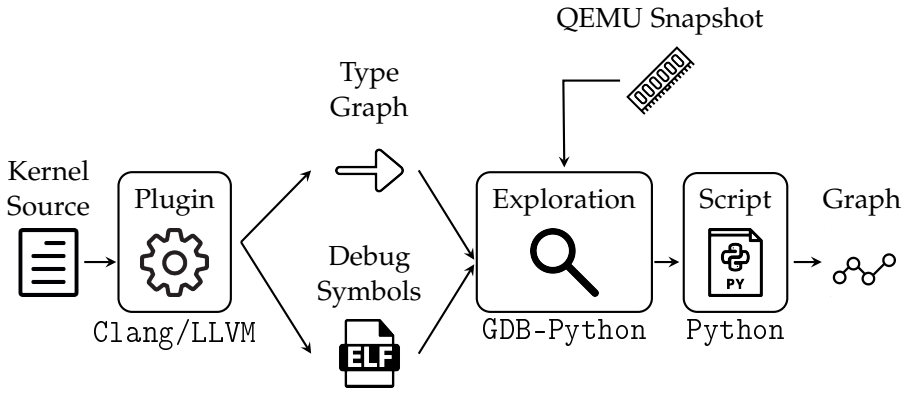


Figure 5.4: System Overview.

However, we want to stress that this limitation does not invalidate our findings since the graph extracted by our approach is *not* incorrect, but only potentially incomplete.

5.4.5 *Implementation*

Our final system is illustrated in Figure 5.4. It consists of an LLVM compiler plugin to perform the *points-to analysis* on the kernel code at compile-time and a set of python `gdb` extensions that combine the information extracted in the previous step with the information provided by kernel debug symbols to identify all kernel objects contained in a memory snapshot acquired using the QEMU emulator. The kernel exploration routine starts by loading a QEMU snapshot, parsing the type graph, and appending the global object symbols to an internal worklist. At this point the real exploration begins: an object is fetched from the worklist and analyzed using the heuristics we adopted to identify invalid or uninitialized memory. If it is well-formed, each of its field are processed to identify structures, pointers to other structures, or arrays of either type. All them are retrieved and appended to the worklist – paying attention to implement the techniques described above to handle abstract data types. These objects are then processed by a separate component responsible

to build the final kernel graph that we will later use to carry out our experiments.

5.4.6 *Final Kernel Graph*

We built our kernel graph using *graph-tool* [Pei14], a python library designed to handle large networks. To reduce the size of the graph, we chose to represent with one vertex each *outer* structure identified during the exploration. In other terms we decided to group together, in a single vertex, all the nested structures (but we keep the nesting information as it is needed when we need to move from the graph space back to the memory analysis heuristics). This transformation also makes the graph directed, and result in only one type of edges that represent pointers from a structure to another. As we will thoroughly discuss in Section 5.5 we assign a number of different weights to each node and edge to allow for several comparisons among different paths.

Figure 5.5 shows a kernel graph counting 109,000 nodes and 846,000 edges, plotted using Gephi [BHJ+09]. This graph contains more than 41,000 strongly connected components with the vast majority (95%) containing only one node. On the other hand, the largest one contains 53% of the vertices and has a diameter of 272 nodes. As we will discuss in Section 5.6, this has important consequences for memory analysis, as it results in a multitude of available paths to move from one node to almost anything else in the kernel memory. The vertex with the highest in-degree is of type `super_block`, pointed by more than 11,000 `inodes` and 11,000 `dentrys`. If we exclude the file system, the node with the highest degree is a `vm_operations_struct`, pointed by more than 4200 `vm_area_structs`.

In the picture, the size of labels and node is adjusted according to the betwenees centrality of a node. This type of centrality counts how many shortest path between every pair of nodes pass through a node. In other terms, the larger the size the more often a node is present inside every shortest path. The node color depends instead on the kernel subsystem the object belongs to. By using the name of the file where the object

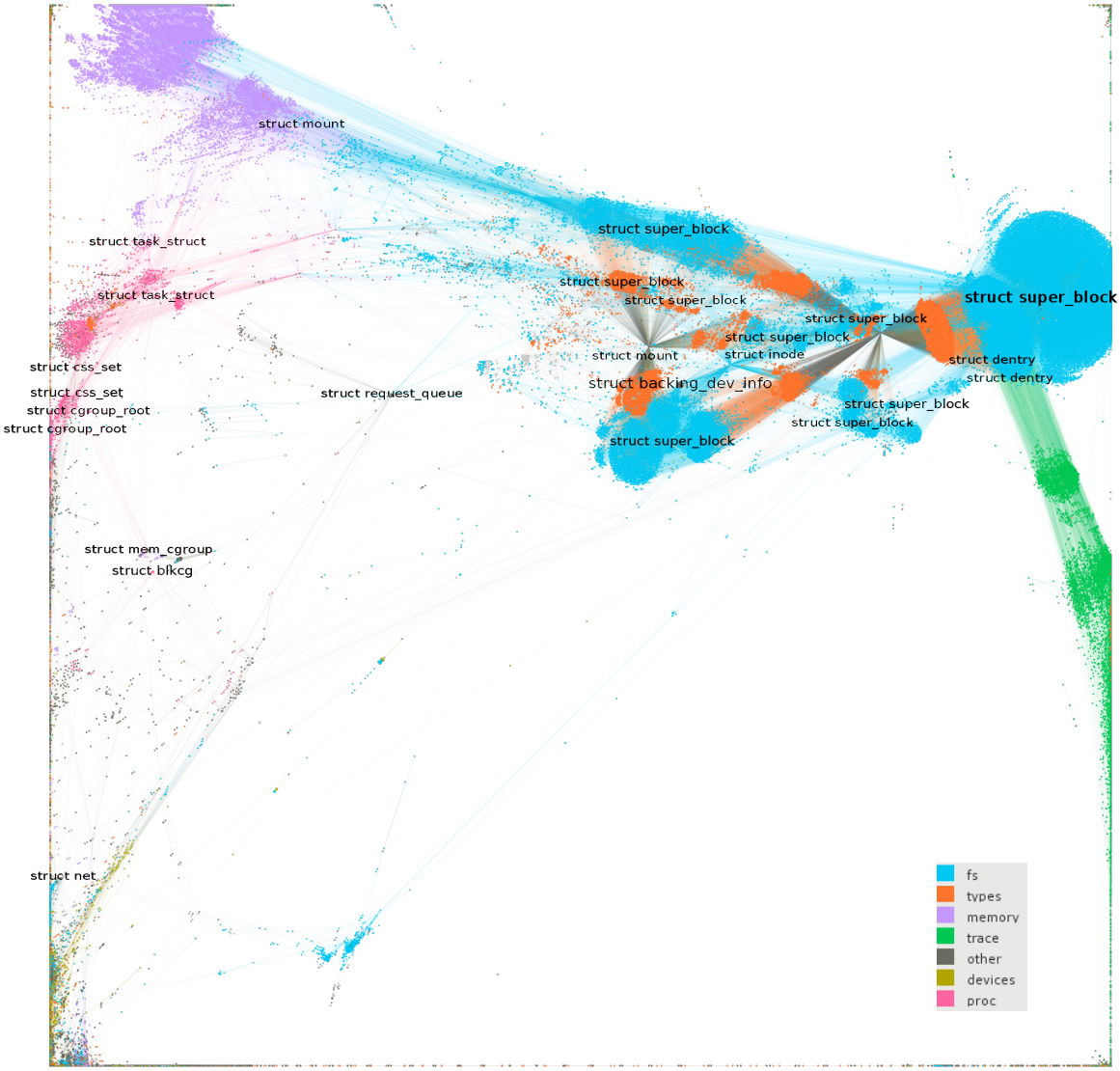


Figure 5.5: Kernel Graph

is defined we were able to classify them in roughly 7 classes, from file system to object related to memory or process management.

5.5 METRICS

In the previous section we described how we extracted a global map of a running kernel that can serve as basis for our analysis. However, without any further information, the only way we can compare two paths on the graph is by looking at their *length*, computed by counting either the total number of nodes or the total number of unique structures that need to be traversed. In fact, this simple approach may resemble the one adopted today by most of the memory forensic tools, where the most straightforward path is often chosen by the developers. However, this solution does not tell anything about the *quality* of a given path, nor about the presence of better options to solve the same problem. To get a solid foundation on which we can compare different techniques we need therefore to define a metric. And since the idea of having an absolute metric is unrealistic, multiple different metrics can be plugged on our graph to study the characteristics of each path.

For our experiments we decided to investigate and add to our graph three numerical and two boolean weights, related to the atomicity, stability, generality, reliability, and consistency of a path. As described below, all of them capture different but important aspects of what an analyst may expect from a memory analysis routine.

Atomicity (numerical)

This weight express the distance in physical memory between two interconnected kernel objects. While this metric is expressed in terms of distance among physical pages, for an easier interpretation we often express it in seconds (as distance in time between the acquisition of the two pages). The atomicity is a very important aspect in most of today's investigation that rely on *non-atomic* dumps. In fact, moving across objects located far apart in memory - and thus acquired far apart on the time scale - can introduce inconsistencies. Intuitively, by using this metric the

best path between a pair of nodes is the one which minimize the time-delta among all visited structures, thus passing only thorough objects acquired very close in time. More precisely, we can adopt three distinct ways to measure Atomicity:

- `Acquisition Window` (AW) – this is the total window that covers all data structures traversed in the path. E.g., one path may walk fifteen objects, all of which were acquired in a period of 23 seconds.

- `Cumulative Time Gap` (CTG) – this is the sum of the time difference of each edge traversed in the path. For instance, if a path visits three consecutive nodes (A, B, and C) and the difference between the acquisition time of the pointer in A and the content of B was 7 seconds and the difference between B and C was 3, the CTG would be 10 seconds.
- `Maximum Time Gap` (MTG) – this just takes into account the longest "jump" in a path. In the previous example, this would be 7 seconds.

All three measures are related to the Atomicity, but they capture different aspects. If it is important than none of the visited structures have changed during the acquisition, AW is the best metric. CTG gives instead a cumulative probability that things can go wrong by following links. The more edges are traversed, and the more far apart are the objects on the end of those edges, the more likely it is than a link can be corrupted due to the non-atomicity of the dump. Finally, MTG provides an estimation of the single most fragile edge in a path. This can be an important information, as traversing 10 edges each one a second apart can be a better option than traversing a single link with a nine seconds delay in the acquisition.

This can lead to some counter-intuitive results. For example, let suppose our graph analysis identifies two paths to reach a certain target structure C namely $\{\texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{C}\}$ and $\{\texttt{A} \rightarrow \texttt{X} \rightarrow \texttt{Y} \rightarrow \texttt{B} \rightarrow \texttt{C}\}$ (for simplicity we ignore the name of the pointers). Both paths start from a structure `A` but the first traverses a single node `B` before reaching the destination while the second takes a detour through two other intermediate

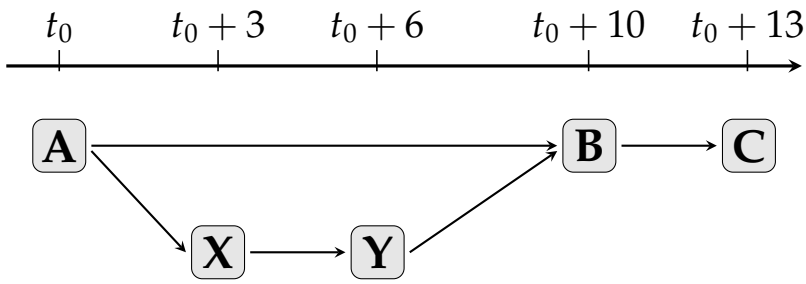


Figure 5.6: Time acquisition of nodes belonging to two paths.

data structures `Y` and `Z` before re-joining the first path. Figure 5.6 shows the two paths on a time scale, that represent at which time the memory containing each data structure was collected.

While the second path is obviously a longer variation of the first, and therefore seems logical to believe that has nothing better to offer, it is very well possible that the detour reduces the probability of incurring in broken links. The pointer `A` $\rightarrow$ `B` was in fact collected 10 seconds before the object `B`, while the longest path decreases these time gaps to a maximum of four seconds. Whether this is an advantage or not depends on how often those pointers are modified in a running kernel, which we capture with our next metric.

Stability (numerical)

This weight expresses the stability over time of a given node or edge on the graph. Some structures are allocated at boot time and are never modified afterwards, while other parts of the graph are very ephemeral and contain structures that get allocated and de-allocated multiple times per second. By computing a heat-map of the stability of each edge (extracted by processing a number of consecutive snapshots), this weight can provide a valuable information on how the kernel map evolves over time, on which paths are more stable, and on which are instead more ephemeral and may only exists for short periods of time.

We measure Stability by computing the `Minimum Constant Time` (MCT) of all links in a path. The MCT can tell, for instance, that over a certain

number of memory images all edges traversed by a certain heuristic remain constant for a minimum time of 30 seconds. In our experiments, we computed this metric by taking a snapshot of the same system at seconds 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 100, 200, 350, 700, 1000, 3000, 5000, 8000 and 12000.

Surprisingly we found that the 81% of edge are stable, i.e., they never change across our experiments. The majority of them are objects related to the file system (`inode` and `dentry`), which the kernel caches for performance reasons. But this does not mean the graph does not evolve, actually quite the opposite. For example, we saw an increase of more than 60% of both nodes and edges between the graph built at $t = 0$ and the one built at $t = 700$.

Moreover, if we exclude the filesystem subsystem and the paths that remained stable over all our experiments, 11% of the edges changed in less than 10 seconds, 12.5% in less than a minute, and 97% in the first hour.

Generality (numerical)

This weight captures another important problem of memory forensics: the constant change in the layout of kernel objects. This is due to several factors. First of all the kernel is always under active development which means that fields are continuously added to and removed from kernel objects definitions. Moreover, the layout is also influenced by the configuration options chosen at compile time. Existing tools mitigate this problem by requiring additional compile-time information (part of what it is normally called an *OS Profile*). Unfortunately, there are cases in which this information is not available, which today greatly complicate (if not completely preclude) the ability of analyzing a particular memory dump. Therefore, it would be interesting to compute analysis paths that traverse structures which change very rarely across different distributions, kernel versions, and enabled kernel options.

For this reason we downloaded 85 kernels from the Ubuntu repository, spanning from version `4.4.0-21` to `4.15.0-20`. For every object defined in each of these kernels we extracted the offset of the fields required

for navigation – such as structure pointers or array of structures. We aggregated this information in a single `Kernels Counter` (KC) weight computed by counting over how many of the 85 kernels an entire path would remain constant (i.e., all its traversed link were present at the same offsets in their corresponding structures).

Reliability (boolean)

This is a very important aspect in memory forensics and captures how tamper-resistant is a given path on the graph, assuming an attacker is capable of reading and writing arbitrary kernel memory. Some paths are very easy for an attacker to modify, and therefore cannot be trusted by an analyst whenever she suspects the attacker might have gained admin privileges on the machine. On the other hand, other paths are more robust, as breaking them would make the system unstable. This can potentially result in programs malfunction or termination and, in the worst case, in a crash of the entire operating system. The robustness of individual data structures has already been studied in the past by several works [BGI08; Dol+09; Pra+13]. But here we are instead interested in the reliability of a path, i.e., not in the fact that individual fields (such as a file name) can be modified, but whether an attacker can tamper with the edges that need to be traversed to prevent a certain heuristic to reach its destination (to the best of our knowledge, this problem has never been addressed in the literature). Being able to compute a path on the graph that only traverses tamper-resistant edges may have a great impact on memory forensics. While today we still do not have enough information to color the entire graph according to this metric, we can still compute the reliability on demand. This means that we cannot compute the optimal solution according to its reliability, but once we have a candidate solution we can perform experiments to verify it.

Consistency (boolean)

As a final property in this list we want to show how metrics can also be aggregated to capture more complex properties of a path. For this example we chose to combine the *stability* and *atomicity* of a path in a

single measure that captures how likely it is for a given path to traverse consistent information. Intuitively, traversing a path whose nodes were acquired over a period of 20 seconds may be acceptable if those structures change very rarely, but completely unacceptable if its links are modified every few milliseconds. We capture this aspect by consider a path *consistent* if and only if the acquisition gap of each edge is lower than the minimum change time of the edge as computed in all our snapshots.

5.6 EXPERIMENTS

We now discuss how our graph-based framework can be used in different scenarios, in which we investigate existing techniques used by Volatility [Wal07], we discuss the intricacies of computing optimal paths, and we discover new solutions to reach all processes running in a system. In any case, these are only examples of what can be achieved by adopting a more systematic approach to memory forensics, and many more applications can benefit from our framework.

All experiments were conducted on a QEMU machine equipped with 2GB of RAM and 4 virtual CPUs, running `wordpress` on top of a `LAMP` stack. Before and between the acquisitions, we generated some activity by visiting the CMS pages and performing basic system administration task, such as logging in via `ssh` and updating the list of packages.

5.6.1 *Scenario 1*

In the first scenario we want to apply our methodology to study the quality of current memory forensics techniques. For our example we selected seventeen Volatility plugins that explore different subsystems (process, network, and filesystem) and mapped them as *paths* in our kernel graph. To achieve this we manually analyzed each plugin and extracted which global variables and kernel objects are traversed. With this information we were able to write a python script which automatically extracts these paths from our graph. Note that many plugins traverse similar kernel

structures (e.g `linux_pslist` and `linux_pstree`) so, to avoid duplicates, we only report results for a subset that rely on different information. The final list of the plugins we analyzed is reported in Table 5.1.

Before looking at the individual metrics, we wanted to investigate to which degree the structures traversed by these heuristics are interconnected. The total number of unique objects used by this heuristics depends on the size of the graph. In our experiments they vary from 20 to hundreds of thousands. As we already introduced in Section 5.4.6, by averaging over the 25 graphs we created, more than 96% of the nodes used by the heuristics belong to a single giant *strongly* connected component that contains on average 53% of all the nodes in the graph. By combining this information with the nodes visited by Volatility, we found that this component contains all the information related to running processes, such as their mapped memory and open files, but also the information related to the `arp` table and the `ttys`. The remaining 4% of the nodes used by the Volatility rules are instead scattered among several other components. The biggest one, which contains only 0.5% of the heuristics nodes, contains the information related to the installed modules and, more in general, to the `kobjects` subsystem. Finally, the rest of the nodes belong to components containing only a single node. These are the nodes representing, for example, the global `pid_hashtable` and its associate `hlist_heads`.

This is an important finding, as it means that the vast majority of the information needed for forensic purposes is interconnected and reachable from one another. Translated in practical terms, the presence of this giant connected component means that is enough to locate a single kernel object to reach all the other interesting ones only by dereferencing pointers. This might be beneficial in scenarios where the position of global kernel objects is not available to the analyst. In such cases, one entry point can often be located by memory carving and then used as starting point for every other analysis.

By looking at the atomicity metrics (columns four-to-six in Table 5.1) the first thing that stands out is that the values for the Acquisition Window (`AW`) and the Maximum Time Gap (`MTG`) are very similar and relatively constant across all commands. After further investigation we discovered

Table 5.1: Comparison of Volatility plugins implemented as paths in our graph

Name	Description	# Nodes	Atomicity			Stability	Generality	Consistency	
			AW	CTG	MTG	MCT	KC	*Fast*	*Slow*
`linux_arp`	Prints the ARP table	13	16.24	53.25	16.24	12,000	50/85	✓	✓
`linux_check_afinfo`	Verifies the function pointers of network protocols	24	16.27	44.55	16.05	700	85/85	✓	✓
`linux_check_creds`	Checks processes that share credential structures	248	16.34	453.92	16.24	2	29/85	✓	✓
`linux_check_fop`	Check file operation structures for rootkit modifications	16099	16.38	142,856.15	16.38	0	29/85	✗	✗
`linux_check_modules`	Compares module list to sysfs info, if available	151	16.27	54.06	16.23	700	85/85	✓	✓
`linux_check_tty`	Checks `tty` devices for hooks	13	16.26	17.52	15.69	30	85/85	✓	✓
`linux_find_file`	Lists and recovers files from memory	14955	16.33	35,627.45	16.32	0	85/85	✗	✗
`linux_ifconfig`	Gathers active interfaces	12	16.25	44.19	16.25	12,000	50/85	✓	✓
`linux_iomem`	Provides output similar to `/proc/iomem`	7	16.70	50.09	16.70	12,000	50/85	✓	✓
`linux_lsmod`	Lists loaded kernel modules	12	16.23	44.27	16.05	700	85/85	✓	✓
`linux_lsof`	Lists file descriptors and their path	821	16.33	19,885.52	16.26	0	29/85	✗	✗
`linux_mount`	Lists mounted fs/devices	495	16.33	8488.13	16.32	10	85/85	✓	✗
`linux_pidhashtable`	Enumerates processes through the PID hash table	469	16.67	451.87	16.67	30	31/85	✓	✗
`linux_proc_maps`	Gathers process memory maps	4722	16.27	2629.19	16.24	0	31/85	✗	✗
`linux_proc_maps_rb`	Gathers process maps through the mappings rb-tree	4722	16.27	3310.69	16.24	0	31/85	✗	✗
`linux_pslist`	Lists active tasks by walking task_struct→task list	124	16.27	189.41	16.24	30	31/85	✓	✓
`linux_threads`	Prints threads of processes	157	16.27	280.68	16.24	30	31/85	✓	✓

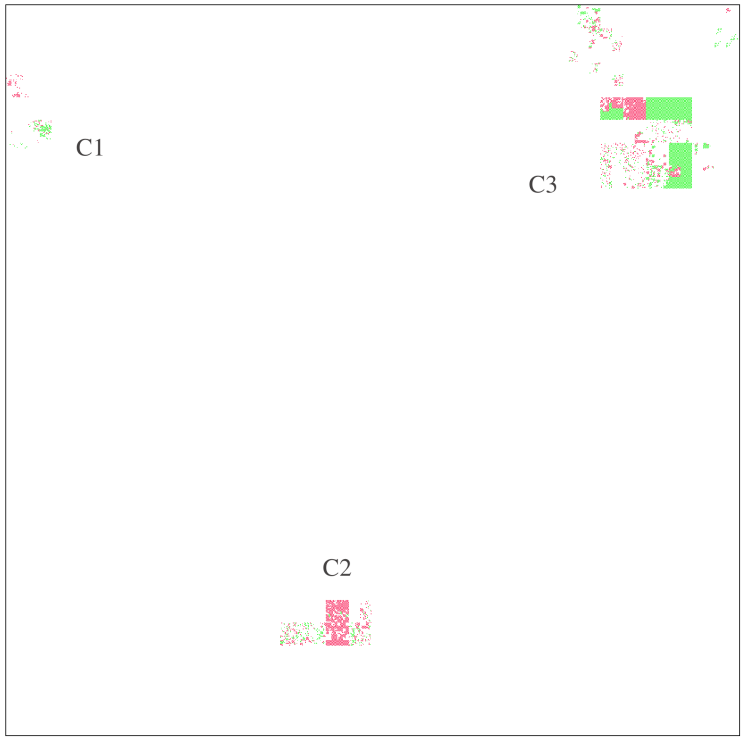


Figure 5.7: View as Hilbert curve of physical memory.

that this is due to the fact that, when compiled with normal configurations, Linux kernel global variables are located in the low part of the physical memory while kernel objects are allocated in the higher end. Since all heuristics start from global symbols, the very first edge already accounts for the maximum gap between two consecutive kernel objects. This also influences the acquisition window, since one of the two farthest objects is always the global variable from where the heuristic starts from. On the other hand, the Cumulative Time Gap (`CTG`) shows more variations as it is also influenced by the number of traversed objects.

To better understand this phenomenon, in Figure 5.7 we plotted the content of the physical memory as a Hilbert curve. In the graph, each pixel represents a physical page and its color shows if the page is traversed

by the Volatility heuristics (red in the graph) or if it contains at least one node of our connected component of the kernel graph (green). It is clear that the relevant data structures are not spread equally on the entire physical memory. Instead, they clearly aggregated around three main clusters, which we marked respectively as C1, C2, and C3. In our experiments the kernel global variables are all located in C1 while other information are often stored in C2 and C3. Therefore, most heuristics start from C1 and then eventually traverse an edge towards one of the other regions - which alone is responsible for the entire `AW` and `MTG` metrics. This physical distribution is also very important for the third scenario presented in Section 5.6.3, where we will encounter heuristics that need to hop back and forth from the three clusters, significantly impacting the atomicity metrics.

The second surprising result of this first scenario is the fact that the Kernel Counters (`KC`) of six plugins *never* changed across all the different kernel versions we used in our analysis. This means that, even when fields were added or removed from these object, the offsets of the fields used by the plugins remained constant. This has important implications for current memory forensics tools where a *profile* of the kernel is needed to analyze a memory dump. Our experiment suggests that, at least for locating certain information, a generic structure layout can be used across almost 100 kernel versions, released as far as 2 years apart.

Another important propriety we evaluated in this first scenario is the consistency of the selected techniques. This is especially useful to better understand how the continuous modifications of kernel objects might impact memory dumps taken in a non-atomic fashion. This was recently listed by Case et al. [CR17] as "*one of the most pressing issues*" of memory forensics. While Case focused on page smearing (an inconsistency between the page tables and the referred physical memory), with this experiment we show that this problem does not affect only page tables but also references among kernel objects. The most important variable that influence the consistency of the memory is the duration of the acquisition process. To align with real world scenarios we run two different tests, by setting the acquisition ratio respectively to the fastest and to the

slowest tool as reported by McDown et al. [McD+16]. In that study, the authors compared seven different memory acquisition tools, chosen from a survey conducted over 41 companies specialized in memory forensics.

Interestingly, out of the 17 plugins we tested, three have a stability of 12,000 seconds, which means that none of the links they traversed *ever* changed over a period of more than three hours. At the other end of the spectrum, eleven plugins walked links that remained stable for less than a minute (and in five cases even less than one second). In this case, this may result in wrong pointers depending on how far in the physical memory were the page containing the link and the page containing the linked object. In fact, the last column of Table 5.1 shows that our analysis found inconsistencies in five (when the fastest tool to acquire the memory was used) or seven (in the case of the slowest solution was used) plugins. The affected plugins interest different parts of the kernel, but they can be divided to three distinct categories: *Memory* (`linux_proc_maps`, `linux_proc_maps_rb`), *File system* (`linux_check_fop`, `linux_find_file`, `linux_lsof`, `linux_mount`) and *Process* (`linux_pidhashtable`)

In the Memory category we found respectively 33 inconsistencies that affected the connections among `vm_area_struct` of a process, which are kept both in a linked list and in a red-black tree. These errors affected five instances of `apache`, one of `systemd-login` and one of `agetty`. The filesystem category included 40 unique inconsistencies in the hierarchy of dentries (fields `d_subdirs` and `d_child`) 53 in the mapping from a `dentry` to an `inode` (field `d_inode`). The latter object was also involved in 43 cases of inconsistency towards its `file_operations` object (field `i_fop`), while 23 `file` object had inconsistent edges pointing to their `dentry` and its `mount` objects. (field `f_path.dentry` and `f_path.mnt`). The most interesting cases of inconsistencies in this category – 10 in total – involved the array containing the pointers to the `files` opened by a process. This array belonged to three distinct instances of `apache`, one of `systemd` and one of the `mysql` database. In the process category, we only detected one case of inconsistent edge between a `struct pid` and the pointed `task_struct`.

To systematically understand if these inconsistent paths can be avoided, we used once again our kernel graph – this time by filtering out *all* the

5,000 inconsistent edges, and searching for alternative paths to reach the same objects used by the affected plugins. Our graph exploration was able to discover alternative paths for 107 out of 213 inconsistent edges. For example, in the case of inconsistent array of opened files for the `systemd` process the alternative path — which traversed 11 additional nodes — was able to reach the target `file` by first locating the `task_struct` of the same process, then accessing its corresponding `files_struct` and from here reaching the `file` via the `fd_array` field (an array only used when the process opens less than 64 files). While these detours were sufficient in our experiments to retrieve the missing information, more experiments are required to understand if those alternative paths can be generalized to other scenarios. In any case, they show once more that the giant connected component that hosts most of the relevant data structures may allow analyst to find alternatives solution to mitigate the presence of wrong pointers and inconsistent information. Sadly, almost 50% of the affected pointers did not allow for an alternative path, thus emphasizing again the severe consequences that the lack of atomicity can have on memory analysis.

5.6.2 *Scenario 2*

In our second case study we want to understand if we can employ the kernel graph to find new heuristics for common forensics tasks. In particular, we focus on the starting point of many forensics investigation: listing the processes running at the acquisition time. Currently Volatility implements three different plugins[1] to list the processes, respectively by walking the process list, by using the `pidhash` hashtable, and by parsing the kernel memory allocator. However, the latter is only applicable if the kernel uses the `SLAB` allocator. Unfortunately, many distributions, such as Ubuntu and Debian, ships by default with the `SLUB` allocator, which is

1 Volatility also includes a plugin to carve `task_struct` objects by using a signature, but this is a parallel approach that does not require exploring memory but relies instead on pattern-matching.

Table 5.2: Comparison between different heuristics used to find processes

Category	Root Node	New	# Nodes	# task_struct	Atomicity			Stability	Generality	Reliability	Consistency
					AW	CTG	MTG	MCT	KC		
scheduling	runqueues	✓	9	4	16.71	20.08	16.70	0.00	34/85	—	✗
	root_task_group	✓	10	4	16.65	21.14	16.27	0.00	18/85	—	✗
cgroup	css_set_table	✓	172	156	16.27	433.32	16.24	10.00	29/85	✗	✗
	cgrp_dfl_root	✓	186	156	16.30	369.10	16.30	10.00	29/85	✗	✓
memory/fs	dentry_hashtable	✓	58383	23	16.31	58120.38	16.30	0.00	36/85	✗	✗
	inode_hashtable	✓	14999	23	16.32	31594.48	16.31	1.00	36/85	✗	✗
workers	wq_workqueues	✓	427	69	16.68	1727.89	16.24	200.00	39/85	✗	✓
process	init_task (linux_pslist)	✗	124	124	16.27	189.41	16.24	30.00	31/85	✗	✓
	init_task (linux_threads)	✗	156	156	16.27	280.68	16.24	30.00	31/85	✗	✓
	pid_hash (linux_pidhash)	✗	469	156	16.67	451.87	16.67	30.00	30/85	✗	✓

not supported by Volatility and which does not keep track of full *slabs* – thus making this technique not applicable anymore.

The main reason for looking for alternative solutions is that previous research already pointed out that rootkits are already capable of removing a process from the process list, but also to unlink a process from the pid hashtable [Lin+11; RJX09; Rhe+10] thus leaving the forensic analyst without a reliable method to list processes. Moreover, as we already discussed in the previous scenario, the lack of atomicity of a memory dump can introduce inconsistencies and result in broken pointers also in the list of running processes. For these reason, it is important to find new ways to locate processes, so that their output can be compared with other techniques to spot inconsistencies or hidden processes.

This scenario is also interesting as it is harder to translate into a graph exploration problem. In fact, since we are looking for techniques to list all (or a part of) the running processes, this is equivalent to a *collection* of, possibly not homogeneous, paths. As a result, listing all processes is not simply equivalent to a path, but more to an *algorithm* to explore the graph.

Our approach to find new heuristics is the following. First, we discarded all the global roots that do not have a path to reach all the `task_structs` in *every* graph we created. As a result, we were left with 621 global roots (out of more than 8000 we started with). Second, we modified the graph to remove the edges already used by known techniques, such as the `tasks` field. This helps removing all those paths that would just find a different way to reach a single process, and then walk the list like the existing plugins already do. While not useless per se, our goal is to find *new* solutions and not variations of the existing ones.

By only considering the *shortest* paths from every root node to every task structure, our system found more than 100 million distinct paths, generated from a set of more than 966,000 sequences of vertices. This is possible because, as we later discovered, the graph contained many parallel edges connecting the same nodes. In fact, by putting things in perspective, on average every sequence of vertices from a root node to a target object generates more than 100 unique paths. The good news is that this makes extremely difficult for attackers to modify all edges

required to completely hide a process. On the other hand though, this also makes very hard the task of identifying interesting patterns in this multitude of options. For simplicity, we first decided to filter out all *similar* edges – i.e., parallel edges that shares the same metrics (and that therefore are equivalent for our purpose). This operation removed more than 300,000 edges, some of which played an important role in the path explosion. For example, many entries of the array `e_cset_node` of the `css_set` object pointed multiple times to the same vertex. After this operation the number of different paths decreased to about 7.5 millions paths.

We then merged similar paths into *templates*, constructed by keeping only the type of the objects present in the path, and by also removing adjacent nodes with the same type (which capture the ↻ link discussed in Section 5.3). Finally, we removed templates that were subset of other templates, resulting in a final set of 4067 path templates.

By manually exploring these options, we soon realized that they belong to only four main families, depending on the kernel subsystem they live in. The first one is related to the *cgroup* subsystem, the second to the *memory* subsystem through the `mm_struct` structure, the third passes through the *work queues* to reach kernel workers, and the last traverses the `struct rq` and follows the `curr` field, a per-cpu runqueue. The results are summarized in Table 5.2.

Unfortunately, there are no alternative paths that can improve the atomicity. In fact, the bulk of the time gap (16.24 seconds) is due to the difference in the acquisition time of the global entry points (located in C1 in Figure 5.7) and the first task structure (located in C2 and C3). However, all these edges are very stable and in only one case (for the `css_set_table`) the value of this first connection ever changed during our memory acquisition.

The memory-based heuristics walked a red-black tree (`i_mmap`) that is very ephemeral and, while exploring it, we found more than 30 edges that could be inconsistent if the memory dump is not taken atomically. A similar problem affects the scheduler, whose structures also contain links that change very rapidly. We observed an interesting phenomenon in the cgroup-related heuristics. The first is inconsistent as it traverses a pointer

with a very large time gap. However, the second avoid this problem by reaching the same `css_set` structures by taking a detour through several intermediate objects which act as a bridge to lower the time gap. This is an example of the counter-intuitive behavior we introduced in Section 5.5 (Figure 5.6), where we predicted that the most direct path might not always be the best in term of consistency. The worker-related approach was the best in terms of stability, consistency, and generality. However, its goal is to list all active kernel workers and therefore this heuristic is unable to capture normal userspace processes. Finally, the two heuristics in the process category, which represent the Volatility plugins `linux_pslist` and `linux_threads`, had both a stability of 30 seconds. This is strange, as several processes should have started during this time frame. However, new processes were all appended to the tail of the process list without altering the intermediate nodes.

To test the Reliability of the heuristics we wrote a kernel module that tries to hide an userspace process by unlinking it from the path required by each heuristic. As a result, each case required a custom hiding technique. For the cgroup heuristics we deleted the processes from the `cg_list` linked list. For the memory we first found every non-anonymous, i.e. backed by a file, `vm_area_structs`. We then delete all this structures from the red black tree rooted in the `inode`, which keeps track of all the `vm_area_struct` which are currently mapping this file. For the first two process heuristics, we removed the process from the process list (by unlinking `task_struct.tasks`), while for the `pid_hash` we removed the `struct pid` from the hashtable. For the workqueue we instead created a custom workqueue and queued a simple work function that mimicked the behavior of the userspace process we used in our test. We then proceeded by unlinking the worker from the linked list rooted at `worker_pool.workers`.

In all the cases our program continued to run without observable side-effects – showing that each path we listed so far can be tampered with by a properly written rootkit. As we also discussed in Section 5.5, we believe that more experiments are needed to improve the assessment of a path's reliability. While it is true that our program continued to run, there can be a multitude of events (e.g. the kernel starting to swap

Table 5.3: Optimal paths compared with Volatility paths

Name	# Nodes	Atomicity			Stability	Generality	Consistency	
		AW	CTG	MTG	MCT	KC	*Fast*	*Slow*
File A – all structures in one cluster								
`Volatility`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-CTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MCT`	4	0.54	0.54	0.54	**12000**	29/85	✓	✓
`Opt-KC`	4	0.01	0.01	0.01	700	29/85	✓	✓
File B – structures located in two clusters								
`Volatility`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-CTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MCT`	11	16.23	72.84	16.21	12000	36/85	✓	✓
`Opt-KC`	8	9.71	46.15	9.71	0	**50/85**	✗	✗
File C – structures located in one cluster, with intermediate steps in the other								
`Volatility`	4	8.73	17.45	8.73	12000	29/85	✓	✓
`Opt-MTG`	3	0.003	0.003	**0.003**	12000	29/85	✓	✓
`Opt-CTG`	3	0.003	**0.003**	0.003	12000	29/85	✓	✓
`Opt-MCT`	3	16.23	82	16.20	12000	36/85	✓	✓
`Opt-KC`	10	9.71	55.56	9.71	0	**50/85**	✗	✗

memory) that might compromise the stability of the altered system.

5.6.3 *Scenario 3*

In the third scenario we show how we can compute optimal paths, with respect to the different metrics we proposed in this work. As running example, we picked this time the problem of finding the files opened by a given process (identified by its `task_struct`).

To run our experiments we collected all the `task_struct` and all the associated `file` objects and analyzed the paths Volatility would take to move from the first to the second. However, we immediately run into a strange behavior, as the metrics were returning very different results for different files. To understand the reason we had to look closer at how the physical pages were assigned to the different kernel objects.

Figure 5.7 explains very well the three classes of behavior we identified in our experiments. Since the clusters (C1, C2 and C3) are located far apart in memory (and therefore they can be acquired far apart in time), whenever a heuristic moves from one structure contained in one cluster to another contained in a different one, it needs to take a "jump" with associated a considerable time gap. If a `task_struct` and all the intermediate objects needed to reach the open files are located inside the same cluster, then time gaps are extremely small and path are always consistent. In this case paths are already optimal and there is no much room for improvement. If they are instead located in two different clusters, then the atomicity increase by almost nine seconds. However, the picture shows that also in this case it is not possible to find better alternatives, as all paths would need to cross the gap between the clusters– incurring in the same penalty. Finally, there are examples in which the `task_struct` and the `file` objects were located in the same cluster, but the intermediate structures traversed by Volatility resided in the other one. In this case the Volatility heuristic needs to jump across clusters twice, incurring twice in the risk of inconsistent links. But in this third case it might be possible to use our graph to find an alternative path that is fully contained in the same cluster.

An example of each of these three cases is shown in Table 5.3, along with the metrics computed on the Volatility heuristic and those computed

on the optimal paths extracted from our graph. Regarding the cumulative time gap (CTG), our insight was correct and only paths belonging to the third category could be considerably improved. In fact, the table shows that from more than 17 seconds in the Volatility case, the optimal path had a CTG of less than 0.01 seconds. Accordingly, also the MTG decreased with the same magnitude. As we discussed in the previous scenario, finding a consistent path for this particular problem is sometimes possible. Indeed, when this is the case, we were able to find a path that remained stable for all our experiments. Interestingly, for the second case, one of the paths with maximum stability has also higher generality than the one used by Volatility but, since it passes through more nodes, it has an higher CTG. On the other hand, maximizing the generality of a path has a serious impact to its consistency and stability. In fact, while we were able to find paths which are constant over 50 kernels, none of them was consistent, independently to the speed of acquisition.

5.7 DISCUSSION AND FUTURE DIRECTIONS

The goal of our work is to provide a principled way to think about memory forensics as a graph-related optimization task. This way of modeling the problem opens the door to a multitude of different possibilities to evaluate and compare existing techniques, design algorithms to compute new alternative solutions, validate the consistency of kernel structures, or propose heuristics customized to different experiments setup and acquired dump.

We tried to discuss some of these opportunities through our experiments, but we are aware that many questions are still open and new research is needed to shed light to each individual use case. For this reason, we decided to release our code and data to other researchers, hoping that this will facilitate new experiments in this field and accelerate new findings based on our methodology.

In this chapter we focused on the analysis of traditional computers. This choice was simply dictated by the fact that this is the area where memory forensics is more mature and for which most of the heuristics

have been designed so far. Nevertheless, we believe that our system could be used to help researchers to better design and implement future forensic frameworks tailored to emerging technologies such as mobile devices and the Internet of Things (IoT).

Main findings: our experiments show that a large part of the kernel graph belongs to a giant connected component. This means there are thousands, or even millions of possible paths that allow an analyst to move from one node to another. It also means that it is very difficult for an attacker to completely hide some piece of information from all possible paths.

Another consequence of the interconnected topology of the graph is that it is hard for an analyst to simply inspect all possible paths, looking for new techniques to implement in memory forensic tools. We tried to do this in our second scenario, and run into a path explosion problem even by considering only all shortest paths. However, this effort allowed us to discover two new promising techniques (one based on `cgroups` and one on `workrqueues`) that can complement those used today by Volatility [2].

Sadly, the problem of finding an optimal path turned out to be very delicate and dependent on multiple factors. In fact, the exact memory layout when the snapshot is acquired may affect the metrics associated to different links (e.g., one path may be optimal for one dump but poor in another). This may suggest that maybe, instead of relaying on a single solution, new techniques should try to explore the graph by following many parallel paths.

Moreover, we are aware that some of the metrics we proposed in this thesis turned out to be ineffective in the evaluation. However, we decided to include them anyway in the paper for two reasons. First, because we did not know in advance that (for example the Maximum Time Gap) would be irrelevant in the analysis of common Linux kernels. This has nothing to do with the heuristic itself, but with the fact that the kernel allocates global variables (entry points) very far from other objects. We believe this fact to be an interesting finding which came as a consequence of applying our framework. Second, while this is true in our experiments,

2 We implemented both as Volatility plugins

it is probably not the case on other operating systems or OS kernels. So, we believe it is still interesting to implement and discuss those ineffective metrics in our framework.

Finally, we want to stress the fact that our main contribution is not the discovery of new technique, but the introduction of a model that can be used to *reason* about memory analysis, explore its complexity, and perform quantitative measurements.

Future Work: In this thesis we discuss a number of metrics an analyst can use to compare different solutions. However, the list is certainly not exhaustive and we expect more to be defined in the future. More work is also needed to understand which metric is better at capturing certain aspects of an investigation.

Reliability is certainly one of the most important characteristic of an analysis technique. Unfortunately, it is also the only one we discussed that cannot be extracted with automated experiments. More research is needed to fill this gap and enable to compute the reliability of a large amount of links among kernel objects.

Finally, to be useful in practice, our prototype should be applied to a larger number of memory dumps taken from different systems. This could help generalize the results and customize the analysis to an environment that resemble the one under investigation.

6 FUTURE WORK

Memory forensics is a constantly evolving field and continuous research coming from academia and industry is essential for this discipline.

Therefore, a first line of future research focuses on keeping the existing tools and techniques up to date with new hardware features and operating systems development. For example, both AMD and Intel have recently announced the support of physical memory encryption in cloud environments – respectively under the name of Secure Encrypted Virtualization (SEV) and Multi-Key Total Memory Encryption (MKTME). To date, it is still unclear how memory acquisition tools will be able to cope with these new technologies. New operating systems features also introduce challenges that must be tackled by forensics practitioners. For example, a compressed in-memory swap has been introduced in the latest version of the Windows kernel. While this means that the analyst is relieved from the burden of acquiring the pagefile from the disk, it also implies that the inner working of the algorithm implementing this feature must be understood to make this information available to forensics frameworks.

Another line of research that can have a substantial impact on forensics investigation concerns Internet of Things devices and networks appliances. For a range of different reasons, memory forensics analyses on these devices is limited to non-existent. First of all, acquiring the volatile memory involves generally more challenges than the environments where memory forensics is nowadays applied. While in this thesis we showed how to automatically extract profiles to analyze devices running variants of the Linux operating system, a large number of devices run proprietary operating systems whom internals are not even public and documented.

While new areas and new technologies are important, we believe that research in this field should also focus on well known problems that affect the existing solutions. For example, in the past few years many *anti-memory forensics* attacks have been presented by researchers [HS12;

Zha+15; Zha+18; Rut07]. To the best of our knowledge, while these attacks have never been spotted in the wild, we believe the community should not be caught off guard but rather harden their tools and techniques in advance to cope with potential evasive behaviors.

The work presented in this thesis also offers several starting points for future research. For example, a recurring topic in this thesis is the study of inconsistencies related to non-atomic acquisition. In fact, the temporal dimension and the kernel graph – discussed respectively in Chapter 3 and Chapter 5 – are proposed as a way to asses the atomicity of a memory dump. Nevertheless, the problem of finding inconsistencies for a generic kernel data structure is still an open problem. Ideally, kernel lists and trees should be checked with rules before being traversed and the analyst should be warned if an inconsistency is detected. While in this thesis we propose an example of this rules, i.e. to check the list and the tree of memory mappings (Section 3.4.2), it is still unclear if is possible to automatically extracting similar heuristics for other data structures.

Finally, in Chapter 5, we propose several scenarios where the graph of kernel structures can be used. One of these scenarios involves finding new heuristics to perform a forensics task. The way this problem was solved was to list all the paths from global variables to each `task_struct`, extract a *path template* and manually check these templates. This strategy was adopted because forensic plugins are not really `paths` on a graph, but more precisely they represent some form of *pattern*, or *algorithm*, to explore the nodes in the graph. In practical terms this means that proving that a global variable is connected to every task does not always correspond to have found a new heuristic to list the processes. New research is needed to design new approaches to explore the graph and mine information and rules in an automated fashion.

Another aspect of our experiments that requires further research is the fact the Reliability of a path required manual intervention to be evaluated. Fully automating this step could help to build memory forensics tools more robust against both current and future malware evasion techniques.

BIBLIOGRAPHY

[- M18] Redhat crash utility - Mailing List. *Using crash with structure layout randomized kernel*. `https://crash-utility.redhat.narkive.com/WZYTWez6/using-crash-with-structure-layout-randomized-kernel`. 2018.

[AD07] Ali Reza Arasteh and Mourad Debbabi. "Forensic memory analysis: From stack and code to execution history". In: *digital investigation* 4 (2007), pp. 114–125.

[Ade06] Frank Adelstein. "Live forensics: diagnosing your system without killing it first". In: *Communications of the ACM* 49.2 (2006), pp. 63–66.

[Al +11] Noora Al Mutawa et al. "Forensic artifacts of Facebook's instant messaging service". In: *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*. IEEE. 2011, pp. 771–776.

[BA18] Manish Bhatt and Irfan Ahmed. "Leveraging relocations in ELF-binaries for Linux kernel version identification". In: *Digital Investigation* 26 (2018), S12–S20.

[BD04] Michael Becher and Maximillian Dornseif. "Feuriges Hacken-Spaß mit Firewire". In: *21C3: Proceedings of the 21st Chaos Communication Congress*. Vol. 10. 2004.

[BD17] Frank Block and Andreas Dewald. "Linux memory forensics: Dissecting the user space process heap". In: *Digital Investigation* 22 (2017), S66–S75.

[BDK05] Michael Becher, Maximillian Dornseif, and Christian N Klein. "FireWire: all your memory are belong to us". In: *Proceedings of CanSecWest* (2005).

[BGI08] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. "Automatic inference and enforcement of kernel data structure invariants". In: *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE. 2008, pp. 77–86.

[Bha+18] Rohit Bhatia et al. ""Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images". In: (2018).

[BHJ+09] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. "Gephi: an open source software for exploring and manipulating networks." In: *Icwsm* 8 (2009), pp. 361–362.

[BP98] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine". In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[Bug18] GCC Bugzilla. *Bug 84052 - Using Randomizing structure layout plugin in linux kernel compilation doesn't generate proper debuginfo*. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84052`. 2018.

[Bur+11] Elie Bursztein et al. "Openconflict: Preventing real time map hacks in online games". In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 506–520.

[Car+09] Martim Carbone et al. "Mapping kernel objects to enable systematic integrity checking". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 555–565.

[Car15] Harlan Carvey. *Digital forensics of the physical memory*. 2015. URL: `http://seclists.org/incidents/2005/Jun/22`.

[Cas+10] Andrew Case et al. "Treasure and tragedy in kmem_cache mining for live forensics investigation". In: *digital investigation* 7 (2010), S41–S47.

[CBS11] Richard Carbone, C Bean, and M Salois. *An in-depth analysis of the cold boot attack: Can it be used for sound forensic memory acquisition?* Tech. rep. DEFENCE RESEARCH and DEVELOPMENT CANADA VALCARTIER (QUEBEC), 2011.

[CG04] Brian D Carrier and Joe Grand. "A hardware-based memory acquisition procedure for digital investigations". In: *Digital Investigation* 1.1 (2004), pp. 50–60.

[Cha+10] Ellick Chan et al. "Forenscope: A framework for live forensics". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM. 2010, pp. 307–316.

[CL16] Juan Caballero and Zhiqiang Lin. "Type inference on executables". In: *ACM Computing Surveys (CSUR)* 48.4 (2016), p. 65.

[CMR10] Andrew Case, Lodovico Marziale, and Golden G Richard III. "Dynamic recreation of kernel data structures for live forensics". In: *Digital Investigation* 7 (2010), S32–S40.

[Coh12] M Cohen. *WinPMEM*. 2012.

[Coh14] Michael Cohen. "Rekall memory forensics framework". In: *DFIR Prague* (2014).

[Cox+18] Guilherme Cox et al. "Secure, Consistent, and High-Performance Memory Snapshotting". In: *Proceedings if the 8th ACM Conference on Data and Application Security and Privacy conference*. 2018.

[Coz+08] Anthony Cozzie et al. "Digging for Data Structures." In: *OSDI*. Vol. 8. 2008, pp. 255–266.

[CR16] Andrew Case and Golden G Richard III. "Detecting objective-C malware through memory forensics". In: *Digital Investigation* 18 (2016), S3–S10.

[CR17] Andrew Case and Golden G Richard. "Memory forensics: The path forward". In: *Digital Investigation* 20 (2017), pp. 23–33.

[Cui+12] Weidong Cui et al. "Tracking Rootkit Footprints with a Practical Memory Analysis System." In: *USENIX Security Symposium*. 2012, pp. 601–615.

[Das00] Manuvir Das. "Unification-based pointer analysis with directional assignments". In: *Acm Sigplan Notices* 35.5 (2000), pp. 35–46.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Dol+09] Brendan Dolan-Gavitt et al. "Robust signatures for kernel data structures". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 566–577.

[Dol+11] Brendan Dolan-Gavitt et al. "Virtuoso: Narrowing the semantic gap in virtual machine introspection". In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 297–312.

[Dol07] Brendan Dolan-Gavitt. "The VAD tree: A process-eye view of physical memory". In: *digital investigation* 4 (2007), pp. 62–64.

[DT08] Brendan Dolan-Gavitt and Patrick Traynor. *Using kernel type graphs to detect dummy structures*. Tech. rep. Technical report, Georgia Tech, 2008.

[Fen+14] Qian Feng et al. "Mace: High-coverage and robust memory analysis for commodity operating systems". In: *Proceedings of the 30th annual computer security applications conference*. ACM. 2014, pp. 196–205.

[Fok+11] Alexander Fokin et al. "SmartDec: approaching C++ decompilation". In: *2011 18th Working Conference on Reverse Engineering*. IEEE. 2011, pp. 347–356.

[GF16] Michael Gruhn and Felix C Freiling. "Evaluating atomicity, and integrity of correct memory acquisition methods". In: *Digital Investigation* 16 (2016), S1–S10.

[GL16] Yufei Gu and Zhiqiang Lin. "Derandomizing kernel address space layout for memory introspection and forensics". In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM. 2016, pp. 62–72.

[GLB13] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. "Hypervisor memory forensics". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2013, pp. 21–40.

[Gor] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. URL: `http://www.makelinux.net/books/lvmm/understand007#toc31`.

[Gra16] Mariano Graziano. *ksfinder - Retrieve exported kernel symbols from physical memory dumps*. `https://github.com/emdel/ksfinder`. 2016.

[Gu+14] Yufei Gu et al. "Multi-aspect, robust, and memory exclusive guest os fingerprinting". In: *IEEE Transactions on Cloud Computing* 2.4 (2014), pp. 380–394.

[Gui17] Adrien Guinet. *wannakey*. `https://github.com/aguinet/wannakey`. 2017.

[Hal+09] J Alex Halderman et al. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[HBN09] Brian Hay, Matt Bishop, and Kara Nance. "Live analysis: Progress and challenges". In: *IEEE Security & Privacy* 2 (2009), pp. 30–37.

[HL07] Ben Hardekopf and Calvin Lin. "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code". In: *ACM SIGPLAN Notices*. Vol. 42. 6. ACM. 2007, pp. 290–299.

[Hof+11] Owen S Hofmann et al. "Ensuring operating system kernel integrity with OSck". In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 279–290.

[HS12] Takahiro Haruyama and Hiroshi Suzuki. "One-byte modification for breaking memory forensic analysis". In: *Black Hat Europe* (2012).

[HT01] Nevin Heintze and Olivier Tardieu. "Ultra-fast aliasing analysis using CLA: A million lines of C code in a second". In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 254–263.

[Hue+07] Ewa Huebner et al. "Persistent systems techniques in forensic acquisition of memory". In: *Digital Investigation* 4.3-4 (2007), pp. 129–137.

[Ibr+13] Amani S Ibrahim et al. "DIGGER: Identifying OS Kernel Objects for Run-time Security Analysis". In: *International Journal on Internet and Distributed Computing Systems* 3.1 (2013), pp. 184–194.

[Jin+14] Wesley Jin et al. "Recovering C++ objects from binaries using interprocedural data-flow analysis". In: *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM. 2014, p. 1.

[Jon07] Ryan Jones. "Safer live forensic acquisition". In: *Computer Science Laboratory, University of Kent at* (2007).

[Kor07] Jesse D Kornblum. "Using every part of the buffalo in Windows memory analysis". In: *Digital Investigation* 4.1 (2007), pp. 24–29.

[Le 18] Stefan Le Berre. *From corrupted memory dump to rootkit detection*. `https://exatrack.com/public/Memdump_NDH_2018.pdf`. 2018.

[Lig+14] Michael Hale Ligh et al. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[Lin+11] Zhiqiang Lin et al. "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures." In: *NDSS*. 2011.

[LK08] Eugene Libster and Jesse D Kornblum. "A proposal for an integrated memory acquisition mechanism". In: *ACM SIGOPS Operating Systems Review* 42.3 (2008), pp. 14–20.

[LPF19] Tobias Latzo, Ralph Palutke, and Felix Freiling. "A universal taxonomy and survey of forensic memory acquisition techniques". In: *Digital Investigation* 28 (2019), pp. 56–69.

[LV08] Marthie Lessing and Basie Von Solms. "Live forensic acquisition as alternative to traditional forensic processes". In: *International Conference on IT Incident Management & IT Forensic*. 2008.

[LWN09] LWN. *Linux kernel design patterns - Part 2*. `https://lwn.net/Articles/336255/`. 2009.

[LZX10] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. "Automatic reverse engineering of data structures from binary execution". In: *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University. 2010, p. 5.

[Mac13] Holger Macht. "Live memory forensics on android with volatility". In: *Friedrich-Alexander University Erlangen-Nuremberg* (2013).

[Man] Mandiant. *Memoryze*.

[Mar+10] Lorenzo Martignoni et al. "Live and Trustworthy Forensic Analysis of Commodity Production Systems." In: *RAID*. Springer. 2010, pp. 297–316.

[Mar17] Jean Marsault. *Volatility-notpetyakeys*. `https://github.com/Iansus/Volatility-notpetyakeys`. 2017.

[Mat+13] Michael Matz et al. "System v application binary interface". In: *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[MC13] Andreas Moser and Michael I Cohen. "Hunting in the enterprise: Forensic triage and incident response". In: *Digital Investigation* 10.2 (2013), pp. 89–98.

[McD+16] Robert J McDown et al. "In-Depth Analysis of Computer Memory Acquisition Software for Forensic Purposes". In: *Journal of forensic sciences* 61 (2016), S110–S116.

[Myc99] Alan Mycroft. "Type-based decompilation (or program reconstruction via type reconstruction)". In: *European Symposium on Programming*. Springer. 1999, pp. 208–223.

[Ots+18] Yuto Otsuki et al. "Building stack traces from memory dump of Windows x64". In: *Digital Investigation* 24 (2018), S101–S110.

[PB19] Fabio Pagani and Davide Balzarotti. "Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 1751–1768. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/pagani`.

[Pei14] Tiago P. Peixoto. "The graph-tool python library". In: *figshare* (2014). DOI: `10.6084/m9.figshare.1164194`. URL: `http://figshare.com/articles/graph_tool/1164194` (visited on 09/10/2014).

[PFB19] Fabio Pagani, Oleksii Fedorov, and Davide Balzarotti. "Introducing the Temporal Dimension to Memory Forensics". In: *ACM Transactions on Privacy and Security (TOPS)* 22.2 (2019), p. 9.

[PH07] Nick L Petroni Jr and Michael Hicks. "Automated detection of persistent kernel control-flow attacks". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 103–115.

[PKH07] David J Pearce, Paul HJ Kelly, and Chris Hankin. "Efficient field-sensitive pointer analysis of C". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.1 (2007), p. 4.

[Pra+13] Aravind Prakash et al. "Manipulating semantic values in kernel data structures: Attack assessments and implications". In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE. 2013, pp. 1–12.

[QV15] Nguyen Anh Quynh and Dang Hoang Vu. *Unicorn-The ultimate CPU emulator*. 2015.

[RAS14] Vassil Roussev, Irfan Ahmed, and Thomas Sires. "Image-based kernel fingerprinting". In: *Digital Investigation* 11 (2014), S13–S21.

[Rei+12] Alessandro Reina et al. "When hardware meets software: a bulletproof solution to forensic memory acquisition". In: *Proceedings of the 28th annual computer security applications conference*. ACM. 2012, pp. 79–88.

[Rhe+10] Junghwan Rhee et al. "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2010, pp. 178–197.

[RJX09] Ryan Riley, Xuxian Jiang, and Dongyan Xu. "Multi-aspect profiling of kernel rootkit behavior". In: *Proceedings of the 4th ACM European conference on Computer systems*. ACM. 2009, pp. 47–60.

[Ruf08] Nicolas Ruff. "Windows memory forensics". In: *Journal in Computer Virology* 4.2 (2008), pp. 83–100.

[Rut07] Joanna Rutkowska. "Beyond the CPU: Defeating hardware based RAM acquisition". In: *Proceedings of BlackHat DC* 2007 (2007).

[Sal+14] Brendan Saltaformaggio et al. "DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse." In: *USENIX Security Symposium*. 2014, pp. 255–269.

[Sal+15a] Brendan Saltaformaggio et al. "GUITAR: Piecing together android app GUIs from memory images". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 120–132.

[Sal+15b] Brendan Saltaformaggio et al. "Vcr: App-agnostic recovery of photographic evidence from android device memory images". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 146–157.

[Sal+16] Brendan Saltaformaggio et al. "Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images." In: *USENIX Security Symposium*. 2016, pp. 1137–1151.

[Sal18] Brendan Saltaformaggio. "Convicted by Memory: Recovering Spatial-Temporal Digital Evidence from Memory Images". In: Atlanta, GA: USENIX Association, 2018.

[SC13] Johannes Stüttgen and Michael Cohen. "Anti-forensic resilient memory acquisition". In: *Digital Investigation* 10 (2013), S105–S115.

[SC16] Arkadiusz Socała and Michael Cohen. "Automatic profile generation for live Linux Memory analysis". In: *Digital Investigation* 16 (2016), S11–S24.

[Sch06] Andreas Schuster. "Searching for processes and threads in Microsoft Windows memory dumps". In: *digital investigation* 3 (2006), pp. 10–16.

[Sch07] Bradley Schatz. "BodySnatcher: Towards reliable volatile memory acquisition by software". In: *digital investigation* 4 (2007), pp. 126–134.

[Sho+16] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[Son+18] Wei Song et al. "DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 606–618.

[Spe06] Bradley Spengler. "Grsecurity". In: *Internet [May 27, 2006]. Available on: http://grsecurity. net/lsm. php* (2006).

[SPE12] Christian Schneider, Jonas Pfoh, and Claudia Eckert. "Bridging the semantic gap through static code analysis". In: *Proceedings of EuroSec* 12 (2012).

[SS10] Matthew Simon and Jill Slay. "Recovery of skype application activity data from physical memory". In: *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*. IEEE. 2010, pp. 283–288.

[SS11] Matthew Phillip Simon and Jill Slay. "Recovery of pidgin chat communication artefacts from physical memory: a pilot test to determine feasibility". In: *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*. IEEE. 2011, pp. 183–188.

[SSB10] Asia Slowinska, Traian Stancescu, and Herbert Bos. "DDE: dynamic data structure excavation." In: *ApSys*. 2010, pp. 13–18.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. "Howard: A Dynamic Excavator for Reverse Engineering Data Structures." In: *NDSS*. 2011.

[Ste96] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, pp. 32–41.

[Sun+15] He Sun et al. "Reliable and Trustworthy Memory Acquisition on Smartphones". In: *IEEE Transactions on Information Forensics and Security* 10.12 (2015), pp. 2547–2561.

[Svi16] Pavel Sviderski. *Universal memory forensic analysis of Android systems*. `https://github.com/psviderski/volatility-android`. 2016.

[Syl12] Joe Sylve. "Lime-linux memory extractor". In: *Proceedings of the 7th ShmooCon conference*. 2012.

[Sym18] ISTR Symantec. *Internet Security Threat Report*. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf`. 2018.

[TDC10] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. "Reconstruction of composite types for decompilation". In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE. 2010, pp. 179–188.

[Urb+14] David Urbina et al. "Sigpath: A memory graph based approach for program data introspection and modification". In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 237–256.

[VF11] Stefan Vömel and Felix C Freiling. "A survey of main memory acquisition and analysis techniques for the windows operating system". In: *Digital Investigation* 8.1 (2011), pp. 3–22.

[VF12] Stefan Vömel and Felix C Freiling. "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition". In: *Digital Investigation* 9.2 (2012), pp. 125–137.

[VS13] Stefan Vömel and Johannes Stüttgen. "An evaluation platform for forensic memory acquisition software". In: *Digital Investigation* 10 (2013), S30–S40.

[VY05] Amit Vasudevan and Ramesh Yerraballi. "Stealth breakpoints". In: *Computer security applications conference, 21st Annual*. IEEE. 2005, 10–pp.

[Wal07] Aaron Walters. *The volatility framework: Volatile memory artifact extraction utility framework*. 2007.

[Wik] Forensics Wiki. *Memory Imaging*. URL: `https://www.forensicswiki.org/wiki/Tools:Memory_Imaging`.

[WL95] Robert P Wilson and Monica S Lam. *Efficient context-sensitive pointer analysis for C programs*. Vol. 30. 6. ACM, 1995.

[WT14] Jake Williams and Alissa Torres. "ADD-Complicating Memory Forensics Through Memory Disarray". In: *ShmooCon, Jan* (2014).

[XCB09] Chaoting Xuan, John A Copeland, and Raheem A Beyah. "Toward Revealing Kernel Malware Behavior in Virtual Execution Environments." In: *RAID*. Vol. 9. Springer. 2009, pp. 304–325.

[Yam+14] Fabian Yamaguchi et al. "Modeling and discovering vulnerabilities with code property graphs". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604.

[Zha+15] Ning Zhang et al. "Now you see me: Hide and seek in physical address space". In: *Proceedings of the 10th ACM symposium on Information, computer and communications security*. ACM. 2015, pp. 321–331.

[Zha+17] Shuhui Zhang et al. "Research on Linux Kernel Version Diversity for Precise Memory Analysis". In: *International Conference of Pioneering Computer Scientists, Engineers and Educators*. Springer. 2017, pp. 373–385.

[Zha+18] Ning Zhang et al. “Memory forensic challenges under misused architectural features”. In: *IEEE Transactions on Information Forensics and Security* 13.9 (2018), pp. 2345–2358.

[ZMW16] Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. “An adaptive approach for Linux memory analysis based on kernel code reconstruction”. In: *EURASIP Journal on Information Security* 2016.1 (2016), p. 14.