THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

A Zero-Knowledge Approach to Memory Forensics

Thèse présentée et soutenue à Biot, le 04/10/2023, par

Andrea Oliveri

Président	Prof. Aurélien Francillon	EURECOM
Rapporteurs	Prof. Andrea Lanzi	University of Milan
	Prof. Stefano Zanero	Polytechnic of Milan
Examinateurs	Dr. Fabio Pagani	Binarly
	Dr. Sarah Zennou	Airbus
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

"To call up a demon you must learn its name. Men dreamed that, once, but now it is real in another way. You know that, Case. Your business is to learn the names of programs, the long formal names, names the owners seek to conceal. True names..." — Rio, Neuromancer [Gib84]

Abstract

The traditional approach to memory forensics heavily relies on a deep understanding of the internal workings of the operating system. With the proliferation of embedded devices, IoT devices, and cloud-hosted virtual machines, there is a growing diversity of operating systems and CPU architectures. Unfortunately, existing forensics tools often lack support for these varied systems, requiring substantial effort to extend their capabilities. When confronted with a memory dump from an uncommon or unknown operating system, analysts are typically limited to primitive tools that can only extract basic information. To extract more comprehensive structural data, analysts often face the arduous task of reverse-engineering the kernel binary, if available. These limitations hinder the applicability of memory forensics on multiple operating systems.

To address this problem, this thesis introduces, for the first time, the concept of "zero-knowledge memory forensics". This approach aims to conduct memory forensics analysis without any prior knowledge of the underlying OS. While profiles, custom rules, and dynamic introspection remain valuable techniques that yield better results when applicable, we argue that our new approach is essential to rapidly expand memory analysis to a wider range of target systems. We begin by quantifying the impact of the memory dumping process and the non-atomicity of memory dumps on the recoverability and consistency of virtual address spaces and kernel data structures. Then, we propose a method to reconstruct the kernel address space in an OS-agnostic manner solely based on hardware configuration information. Furthermore, we show that it is possible to identify kernel pointers and reconstruct in-memory kernel data structures such as lists and trees using only OS-agnostic properties related to their topology.

Résumé

L'approche traditionnelle de l'analyse forensique de la mémoire volatile repose fortement sur une compréhension approfondie du fonctionnement interne du système d'exploitation. Avec la prolifération des dispositifs embarqués, des appareils IoT et des machines virtuelles hébergées dans le cloud, il existe une diversité croissante de systèmes d'exploitation et d'architectures de processeurs. Malheureusement, les outils de forensique existants manquent souvent de support pour ces systèmes variés, ce qui nécessite des efforts considérables pour étendre leurs capacités. Lorsqu'ils sont confrontés à un dump de mémoire provenant d'un système d'exploitation peu commun ou inconnu, les analystes sont généralement limités à des outils rudimentaires qui ne peuvent extraire que des informations de base. Pour obtenir des données structurales plus complètes, les analystes doivent souvent se livrer à la tâche ardue de la rétro-ingénierie du noyau, si celui-ci est disponible. Ces limitations entravent l'applicabilité de l'analyse forensique de la mémoire volatile sur plusieurs systèmes d'exploitation.

Pour résoudre ce problème, cette thèse introduit, pour la première fois, le concept de la "l'analyse forensique de la mémoire à connaissance nulle". Cette approche vise à réaliser une analyse de la mémoire forensique sans aucune connaissance préalable du système d'exploitation sous-jacent. Bien que les profils, les règles personnalisées et l'inspection dynamique restent des techniques précieuses qui donnent de meilleurs résultats lorsque cela est possible, nous soutenons que notre nouvelle approche est essentielle pour étendre rapidement l'analyse de la mémoire à un plus large éventail de systèmes cibles. Nous commençons par quantifier l'impact du processus de capture de mémoire et de la non-atomicité des captures de mémoire sur la récupérabilité et la cohérence des espaces d'adressage virtuels et des structures de données du noyau. Ensuite, nous proposons une méthode pour reconstruire l'espace d'adressage du noyau de manière indépendante du système d'exploitation, uniquement basée sur les informations de configuration matérielle. De plus, nous démontrons qu'il est possible d'identifier les pointeurs du noyau et de reconstruire des structures de données du noyau en mémoire, telles que des listes et des arbres, en utilisant uniquement des propriétés indépendantes du système d'exploitation liées à leur topologie.

Contents

List of Figures

Chapter 1

Introduction

We live in an era of information technology, where digital devices are essential for our daily activities. Computers, mobile phones, networks and infrastructure devices, embedded systems and even connected home appliances have become ubiquitous in modern society playing a vital role. We rely on these devices to store, process, and transmit our personal and professional data, make payments, and we even entrust them with our safety for example during a flight or using a self-driving car. However, these devices can be the target of malicious actors who seek to compromise them for their own benefit. These actors can gain unauthorized access to these devices and manipulate their content or functionality which can have serious consequences for individuals, organizations, and society at large producing financial loss, national security breaches or even putting people's lives at risk.

Since no system can be completely secure from cyberattacks, it is not enough to rely on preventive defenses alone. It is also necessary to detect and respond to such incidents in a timely and effective way reconstructing the sequence of events that the attacker carried out. This involves identifying the vulnerabilities exploited, the actions performed, and the impact caused by the attack, as well as recovering the artifacts used by the attacker. These are all part of the digital forensics discipline which involves applying scientific methods and techniques to collect and examine digital evidence in a forensically sound manner. One of the earliest examples of digital forensics was focused on the analysis of logs and the status of compromised live systems. This was the case in 1989, when Clifford Stoll, an astronomer and system administrator at Lawrence Berkeley National Laboratory, discovered a West German spy who had infiltrated his network and was stealing data and selling it to the Soviet Union [Sto89].

Digital forensics has undergone significant advancements in the 90s and 00s, encompassing not only the analysis of system logs but also disk and network forensics. Disk forensics involves the extraction of raw data from non-volatile devices, such as hard drives, to identify active, modified, or deleted files. Additionally, it seeks out hidden or encrypted data that can provide valuable insights into the user's activities, preferences, communications, and intentions. On the other hand, network forensics revolves around the capture and analysis of network packets to reconstruct events, detect potential intrusions, and uncover evidence of malicious activities. By scrutinizing network traffic patterns, specialists in network forensics gain crucial insights into the actions undertaken by both attackers and victims, facilitating the reconstruction of events and a comprehensive understanding of the security breach's scope.

In recent years, the realm of digital forensics has extended its reach to include the realms of mobile and cloud computing. Mobile devices, including smartphones, tablets, and wearable

devices, now store a vast amount of personal and sensitive data, holding relevance for various purposes. In this ever-changing landscape, memory forensics emerges as a critical component of digital investigations.

1.1 Memory Forensics

Memory forensics, in particular, is the branch of digital forensics that deals with the analysis of the volatile memory of a computer system. It emerged as a research field, in response to the increasing sophistication and stealthiness of malware and cyberattacks when disk forensics techniques were not sufficient to detect and analyze malware that did not leave traces on non-volatile storage devices.

Volatile memory, also known as random access memory (RAM), is the main memory of a system and is used by the operating system (OS) and user space applications to temporarily store and process data. Together with user space application and kernel code, the RAM contains kernel data structures that maintain information about the state of the system and can be analyzed to extract runtime artifacts such as the list of running processes, loaded kernel modules, network connections, encryption keys, malware signatures, and more. These artifacts can provide valuable information for forensic investigations and incident response, as they can reveal the state and the activity of the system at the time of memory acquisition. A memory forensics investigation consists of three main phases: acquisition, interpretation and analysis.

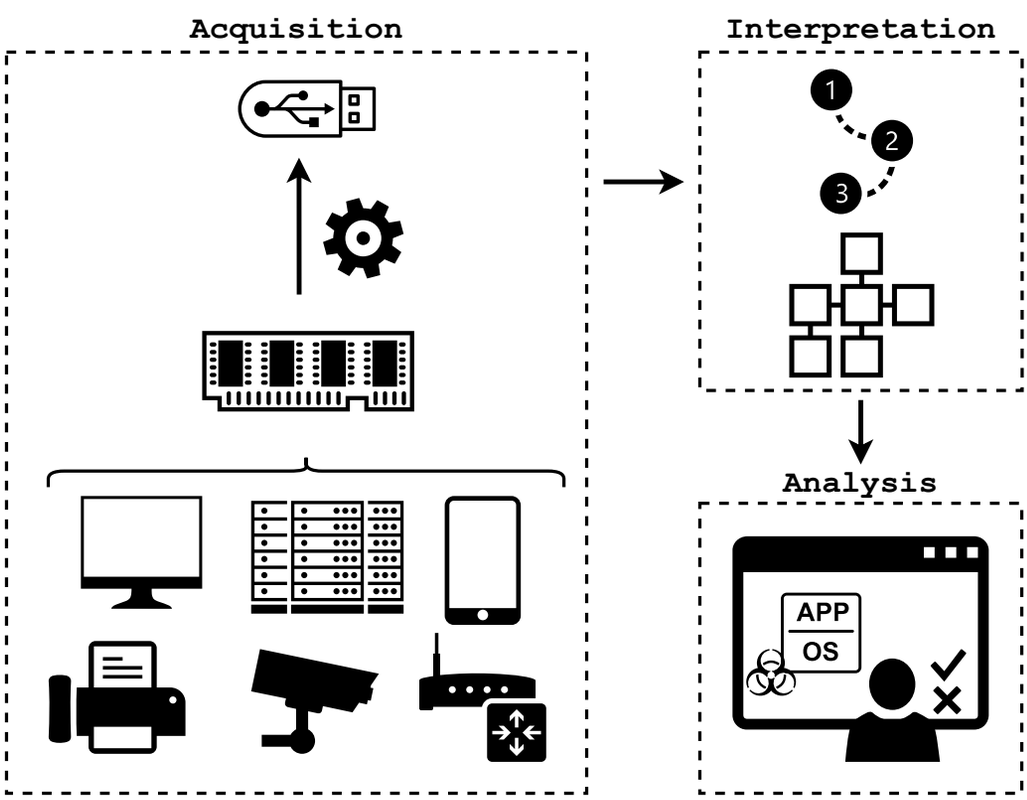


Figure 1.1: Three main phases of a memory forensics investigation: acquisition of the main memory, interpretation of data in the memory dump and analysis of the artifacts.

- **Acquisition** In this phase the contents of the memory is acquired from the target system using specialized tools and techniques. The acquisition process must be done in a forensically sound manner, which means that it must not alter or damage the original memory data, and it must preserve the integrity and authenticity of the evidence.
- **Interpretation** This phase involves identifying and extracting relevant artifacts from the memory image file using a forensic tool or framework. The interpretation process requires a deep knowledge of the data structures and formats used by the operating system and the applications to store information in memory.
- **Analysis** The analyst examines and correlates the forensic artifacts to reconstruct the events and activities that occurred on the target system. This phase requires a combination of technical skills and domain knowledge to interpret and evaluate the evidence.

Two peculiar characteristics of memory forensics pose many challenges to the security researchers and forenics analysts: volatile memory is transient and its content is continuously modified by the operating system and user applications and, unlike a dump of an hard-disk, where data is organized by a well-recognizable file system structure, data in a memory dump lacks clear boundaries and semantics.

1.2 Open Problems in Memory Forensics

Consider the information about the running processes on a system: when the kernel launches a new process it has to maintain the process' name, its memory layout, the loaded libraries, and other metadata to correctly track the process during its execution and be able to free the allocated resources when it is terminated. An analyst that wants to retrieve this information from a memory dump has, as first thing, to collect the content of the memory saving it on a non-volatile storage (acquisition phase).

This task, which on the surface seems trivial, turns out to be full of pitfalls and hidden problems. First of all, different types of machines and CPU architectures require different techniques to perform the dump, and each of them is a harbinger of additional problems. Consider, for example, the case where the analyst has direct access to the machine's hardware: in this scenario one could use debug interfaces on the main board (such as the JTAG interface [RALJA21]) that allow to stop the CPU and read the contents of physical memory directly. However, often these interfaces are disabled by the manufacturers, such as in the case of most smartphones, or are even not accessible except through exploits or require permission levels possessed only by the CPU manufacturer (as in the case of DCI interfaces on INTEL platforms [LSF21]). However, there are other hardware access routes that could be exploited to obtain a dump such as using baseboard management controllers (BMCs) (where present) [LBF20], or using features built into system firmware such as with Firmware Assisted Dumps in the case of IBM POWER platforms [pow23]. If the system has been designed with the idea to dump memory in case of need, it is possible that a PCI-E interface that permits an external and direct access to RAM memory has been installed, or the permission to perform DMA through USB4/Thunderbolt [LPF19] has been enabled at boot time. In almost all cases, however, it's not possible to take a memory dump via hardware and the analyst has to use software with a security privilege level that allows to read all of the system's physical memory. This capture software can be loaded by the

analyst at the time of the dump such as a kernel module, or be already present in the system as in the case of SMM [RFP+12] or EFI modules [LHKF21]. However, the software acquisition method introduces two very subtle problems that can irremediably compromise the dump. The first has to do with using a software component that resides in and uses RAM memory for its operation by altering the contents of the memory itself (a problem equivalent to the observer effect in quantum mechanics). The second problem, has to do with dump atomicity: the dump process is not instantaneous and some amount of time is required to copy each physical page of memory to a non-volatile storage or send it over the network. Most dump tools (except for the usage of hypervisor capabilities in case of memory dump of virtual machines that are not using VM encryption technologies like AMD SEV) do not halt the kernel and user processes during the dump leaving them free to continue to write to and modify the memory. Dumps obtained in this way, present regions of memory dumped at different times that contain pointers to data that have since been changed or data no longer present at the time of the dump or even point to different data structures that randomly occupy the same portion of memory.

After the acquisition, the analyst has to deal with a blob of data that represents the content of the physical memory of the system because, unlike a dump of an hard-disk, where data is organized by a well-recognizable file system structure, data in a memory dump lacks clear boundaries and semantics. To correlate the artifacts contained in the physical memory dump (analysis phase) the analyst has first of all to reveal them by resolving two different semantic gaps (interpretation phase): virtual to physical addresses translation, which deals with translating pointers expressed as virtual addresses to their physical position in the memory, and the detection of kernel data structures e.g. those related to running processes, memory management, and kernel modules. To solve these two problems an analyst must know the OS kernel internals. For well known operating systems like Windows, Linux and macOS, classic forensics tools like Volatility [Wal17] and Rekall [Coh14] have, for each supported OS, collections of *profiles*. A *profile* is a sort of "atlas" which describe, for a specific version and configuration of a specific OS the location of kernel data structures in memory, their size and the type and location of each fields in them. Thanks to *profiles*, using heuristics and hand-tailored OS-specific rules, these tools can recover the virtual to physical mapping and extract relevant forensics information from the analyzed memory dump. The collection of *profiles*, however, needs to be constantly updated because data structures used by an operating system can change as it evolves. Furthermore, for OSs like Linux, which allow a high degree of personalization of the kernel configuration, may even be necessary to create a specific profile even for the single machine that the analyst wants to investigate. Some recent works [PB21, QQY22, FHA+22] have partially overcome the problem of the necessity of specific *profiles* for custom Linux kernel configurations, reconstructing them directly from the memory dumps or performing a forensics analysis without them. However, also in these works, the authors suppose to have a deep knowledge of the kernel internals and be able to identify specific functions able to provide hints on the data structures shapes.

The classical approach to memory forensics is strictly bound to the knowledge of the internals of the OS and this poses a serious challenge for the future of memory analysis. In fact, the rapid increase of embedded devices, IoT ones, and cloud-hosted VMs translates into a more variegated number of operating systems and CPU architectures, which in general are not supported by current forensics tools and require a great effort to be extended. Nowadays an analyst that runs into a memory dump of an uncommon, or worse, unknown operating system, can rely only on primitives tools to carve, for example, raw strings, and have to perform a deep

work of reverse-engineering of the kernel binary (where available) to be able to extract more structural data. For all the above reasons, memory forensics of not common operating systems, for which the virtual to physical address translation and/or kernel data structure organization is not known or no *profiles* are available results impossible showing all the limitations of this approach thereby preventing its applicability on multiple OSs.

1.3 Research Questions

To overcome this problem, we introduce, for the first time, the concept of *zero-knowledge memory forensics*: perform memory forensics analysis without *any* knowledge of the underlying OS. While profiles, custom rules, and dynamic introspection remain valuable solutions that are likely to provide better results when they can be applied, we believe that our new approach is needed to quickly extend memory analysis to a broader class of target systems. Supposing to have taken a memory dump of the unknown OS, using only information derived from the hardware configuration of the machine, we assert that is possible to reconstruct the kernel address space in an OS-agnostic way, and then identify kernel pointers. Starting from them, we assert also that is possible to reconstruct in-memory kernel data structures like lists and trees using only OS-agnostic properties related to their topology. However, to prove the feasibility of our theory we need to answer three questions:

Question #1: Is it possible to *quantify* how much the dump process and the non-atomicity of memory dumps affect the recoverability and consistency of the virtual address spaces and kernel data structures?

As pointed out in previous sections, the most used software-based memory imaging tools share a problem: they do not stop the system while acquiring the memory. This leads to a non-atomic acquisition that has inconsistencies because the dump is not a unitary representation of the memory content at a specific time, but rather resembles a long exposure photo of it.

In the case of an unknown operating system, for which we do not know anything including the way it uses physical memory, the most simple and natural way to dump it is to dump physical pages sequentially according to their physical address. Incidentally, this is also the strategy used by open-source dump tools available for major OSs (such as LiME [Ben23] for Linux or Winpmem [win23] for Windows).

Several research papers have described the non-atomicity problem [VF12, GF16, PFB19], but the footprint on the RAM of the dumping tool itself and the effects of the non-atomicity on the availability, accessibility, and consistency of virtual address spaces and kernel data structures are never been quantified.

Question #2: Is it possible to reconstruct the virtual-to-physical address mapping without *any* knowledge about the operating system, starting only from its memory dump and information about the hardware?

The first step in the identification phase is the translation of virtual to physical addresses. If the OS is known, there are, as mentioned above, heuristics that permit to derive the mapping between addresses. However, if the operating system is unknown these heuristics are not available and commonly used forensics tools such as Volatility [Wal17] are useless. Furthermore,

the problem of reconstructing address spaces in the literature is given as "solved a priori" without ever going into the details of the problem. In the case of an unknown operating system, one must rely only on the machine hardware (which is assumed to be known and documented) since its characteristics do not depend on the operating system. Also in this case the literature is lacking since the few cases that deal with the problem of address space reconstruction using hardware information such as [SG10] are always limited to the x86 architecture ignoring CPU architectures with different virtual address space translation mechanisms even in the case of known operating systems.

Question #3: Is it possible to extract kernel data structures like linked-lists, arrays etc. without *any* knowledge of the operating system, starting only from its memory dump and information about the hardware?

The second and last step in the identification phase is the recognition of kernel data structure that contains forensics-relevant data. For a well-known operating system, an analyst can use a *profile* to instruct forensics tools to locate and explore kernel data structures. It thus turns out that in the case of an unknown operating system, the *profile* approach is not applicable and the analyst is completely blind. In literature, some works have tried to overcome the need for *profiles* using for example instrumented virtual machines running the OS under analysis [DGSTG09, LZX10] to derive information about data structure used by the kernel or analyzing its source code [CMR10, FPW+16, LRW+12, LRZ+11]. All these techniques, however, assume to have some *"auxiliary"* data about the OS or special capabilities over it such as the access to the source code or the possibility to run/rehost it on a VM. We want, instead, to derive forensics-relevant kernel data structures only from the memory dump of the machine without any other information about the OS.

1.4 Contributions

"Is The End of Memory Acquisition As We Know It?"

In this work, we present a system based on record-replay emulation system PANDA [DGHH+15] to track all write operations performed by the OS kernel during a memory acquisition process. This allows us to quantify, for the first time, the number and type of inconsistencies observed in non-atomic memory dumps. We examine the activity of three different operating systems in idle and how they manage physical memory. Then, focusing on Linux, we quantify how different dump modes, file systems, and hardware targets influence the frequency of kernel writes during the dump. We also analyze the impact of inconsistencies on the reconstruction of page tables and major kernel data structures utilized by Volatility to extract forensic artifacts. Our results show that inconsistencies are common and can undermine the reliability and validity of memory forensics analysis of dumps taken in non-atomic ways.
This project was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement N°771844 (BitCrumbs). The tool developed in this work is released as open-source project together with the dataset.

This contribution is currently under submission for peer-review to ACM Transactions on Privacy and Security (TOPS)

"In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics"

In this work, we study for the first time at all how to bridge the semantic gap in virtual-to-physical translations in 10 different CPU architectures in an OS-agnostic way. In each case, we study the inviolable constraints imposed by the memory management unit that can be used to build signatures to recover the required data structures from memory without *any* knowledge about the running OS. We also introduce a static code analysis step to recover the state of the MMU registers configured by the OS at boot time. We build a proof-of-concept tool, MMUShell, that in contrast to the existing forensics tools available, uses only parameters derived from the CPU ISA to recover virtual to physical mappings. We experiment with the extraction of virtual address spaces showing the challenges of performing an OS-agnostic virtual-to-physical address translation in real-world scenarios. We conduct experiments on a large set of 26 different OSs and a use case on a real hardware device. Finally, we show a possible usage of our technique to recover and analyze user space processes running on an unknown OS without any knowledge of its internals permitting an analyst to start a forensics analysis on the system.
This project was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement N°771844 (BitCrumbs) and by the European Unions Horizon 2020 research and innovation programme under grant agreement N°786669 (ReAct). The tool developed in this work is released as open-source project together with the part of the dataset not covered by particular OS license restrictions.

This contribution has been published in an article appeared in ACM Transactions on Privacy and Security (TOPS) Volume 22, Issue 4, July 2022 [OB22]

"An OS-Agnostic Approach to Memory Forensics"

In this work, we introduce the new concept of zero-knowledge memory forensics, which is based on techniques that can recover certain forensic information without any knowledge of the internals of the underlying OS. Our approach allows to automatically identify different types of data structures by using only their topological constraints and then supports two modes of investigation. In the first, it allows to traverse the recovered structures by starting from predetermined *seeds*, i.e., pieces of forensics-relevant information (such as a process name or an IP address) that an analyst knows a priori or that can be easily identified in the dump. We have implemented our technique in a proof-of-concept tool, Fossil, which has permitted us to conduct various experiments in real-world scenarios. Our experiments show that even a single seed can be sufficient to recover the entire list of processes and other important forensics data structures in dumps obtained from 14 different OSs, without any knowledge of the underlying kernels. In the second mode of operation, our system requires no seed but instead uses a set of heuristics to rank all memory data structures and present to the analysts only the most 'promising' ones. Even in this case, our experiments show that an analyst can use our approach to easily identify forensics-relevant structured information in a truly OS-agnostic scenario.
This project was supported by the European Research Council (ERC) under the European

Unions Horizon 2020 research and innovation programme under grant agreement N°771844 (BitCrumbs). The tool developed in this work is relased as an open-source project together with the part of the dataset not covered by particular OS license restrictions.

This contribution has been presented at 30th Network and Distributed System Security Symposium (NDSS) 2023 [ODB23]

Chapter 2

Related Works

Significant research has been conducted in memory forensics in the last decade, resulting in the publication of various papers describing the process of capturing and examining the volatile storage of a target machine in different scenarios. Researchers have tried to answer different questions, such as the possibility to use non-standard acquisition methods to increase the atomicity of the dump [Sch07, HSH+09, MFPC10, RFP+12, LHKF21], the possibility to extract information about the kernel version, the running processes and their address spaces from a memory dump [RAS14, BA18, GL16, SG10] and the identification of kernel data structures in the lack of a valid profile using static analysis on the kernel code or observing its behaviour [DGSTG09, SYLS18, PB21, FHA+22].

Many of the techniques developed by the research community, however, are specific to particular operating system versions and CPU architectures, or they only function under specific conditions. Additionally, the reliability of the generated results can vary depending on the technology employed. Consequently, it is crucial to understand the capabilities and limitations of the respective solutions to effectively retrieve evidence. In the context of this thesis, we will focus in particular on the problems arising from the non-atomicity of dumps and the semantic gap reconstruction.

2.1 Quantification of Inconsistencies in Non-Atomic Memory Dumps

Over time, numerous studies [Kor07, LK08, HBN09, MC13] have pointed out how the lack of atomicity in memory dumps can lead to problems and errors during a forensic analysis. The first to formalize the concept of atomic dumps were, in 2010, *Vomel and Freiling* [VF12] that have also introduced the distinction between integrity correctness and atomicity of a dump. In a later work, *Gruhn and Freiling* [GF16] have evaluated 12 different acquisition tools that adhered to these three criteria. In 2021 *Freiling et al.* have extended the concept of atomicity and integrity also to disk snapshot in [FG21]. More recently, Case and Richard highlighted the pressing issue of page smearing [CRI17]. With the prevalence of servers equipped with large amounts of RAM causing longer acquisition times, the smearing effect has become increasingly common. In 2019, *Pagani et al.* [PFB19] have introduced "temporal dimension" in memory forensics. This dimension provides the analyst with an initial means to evaluate the atomicity of the data structures employed during an analysis. Additionally, a series of experiments were

conducted to demonstrate that page smearing not only introduces inconsistencies in page tables but also indiscriminately impacts any analysis involving structures located in user space.

Among the first to propose a solution to the problem of non-atomicity of memory dumps, even before the formalization of the definition, were *Huebner et al.* who in their work [HBHW07] propose an automatic acquiring of the state of the kernel and user applications periodically performed by the operating system itself. This solution, however, requires a redesign of the internals of the kernel which is not immediately applicable to existing OSs.

An interesting approach to obtaining an atomic snapshot through software acquisition is developed by *Schatza and Bradley* [Sch07]. At runtime, they inject a minimal kernel that stops the execution of the running OS and dumps its memory. This approach, as underlined by the authors, have several limitations due to the strong integration between the dumping kernel and the original OS and the variability of the hardware that must be supported.

In 2009, an approach based on the data remanence effect in memory chips after the reboot of the system is used by Forenscope [HSH+09] to acquire the memory in an atomic way. This technique, known as cold boot memory acquisition has been proven, unfortunately, that strongly depends on the chipset and memory modules used, with certain combinations of components failing to retain RAM content upon reboot [CBS11].

The next year *Martignoni et al.* have introduced HyperSleuth [MFPC10] which uses a custom hypervisor injected at runtime to perform memory dumps using dump-on-write and dump-on-idle in order to be resistant to evasive malware, that can tamper the OS kernel, and obtain an atomic dump at the same time. This promising technology, however, requires activation of virtualization support on the CPU prior to booting the machine itself, and also is not applicable in cases where the operating system itself acts as a hypervisor such as Windows 10 virtual secure mode.

Recently, in *"Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots"* [FHA+22] the authors have collaterally introduced a Linux kernel module that allows atomic memory dumps. However, the developed technique, by temporarily blocking system interrupts, can cause multiple, chain-like crashes of the kernel modules responsible for hardware management at the end of the dump process.

Other solutions developed to obtain atomic dumps involve using modified firmware to run the dump tool at higher privilege levels (as SMM module [RFP+12]) or with the operating system no longer running but with the memory contents still available (as an EFI module [LHKF21]). These two methods, however, require deep modification to the system firmware residing on the motherboard or to be installed in the EFI partition and a valid cryptographic signature to permit to the dump tool to run in SecureBoot environment which is now a default condition for modern operative systems on Intel CPUs.

2.2 Virtual Address Spaces Reconstruction and Data Structure Identification

OS-agnostic data structure recovery, as a particular application of the more general problem of data reconstruction was explored by various authors by using different techniques. Some of these techniques are derived from traditional disk forensics. A classic example is data carving, in which the analyst tries to recover information from the memory dump by using signatures that uniquely identify the data of interest. For instance, *Beverly et al.* [BGC11] create a set of

signatures to carve network packets from a memory dump while *Van Barr* [vAv08] developed a technique to carve memory-mapped files from Windows dumps. A more generic approach is proposed by *Wagner et al.* [WRG15], who reconstructed relational database data structures in memory dumps for different SQL engines.

Saur and Grizzard [SG10] are the first to propose an approach to reconstruct radix trees for Intel x86-32 from both Windows XP and Linux memory dumps. Their parsing technique is based on rules derived from the ISA specifications and tailor-made heuristics derived from the OSs internals, which allows the authors to retrieve virtual and physical address spaces of hidden processes running on a sampled system. Their approach, however, is limited by the non-universality of their parsing rules, which contains heuristics based on how Linux and Windows kernels use page tables, making it impossible to use on dumps of different OSs. A work most closely related to our study is the parallel paper from *Tran-Quoc et al.* [TQHMT21]. The paper focuses mostly on page table reconstruction for Intel x86 only, but the authors also describe a simple use case in which they use the recovered pointers to identify the list of running processes in Linux, BSD, and MS Windows. While the idea to identify and follow pointers is similar to our technique, this work does not discuss data structure reconstruction, cannot deal with trees, arrays, or cases in which the process name is not part of the linked list structure itself, not considering also seed-less introspection. A work more similar to ours for its focus on CPU ISA but with a different goal is [GLB13]. In this work, the authors use OS-agnostic parsing rules to detect, in memory dumps, virtual machine control structures, used by the Intel CPUs to maintain the state of virtual machines.

The more general problem of extracting data structures from memory dumps has already been treated in several previous studies, even if with different goals. For instance, *Dolan-Gavitt et al.* [DGSTG09] have proposed a signature-based approach to define invariants of kernel data structures and generate signatures for their recovery in memory dumps. A similar approach, but based also on probabilistic inferences, was developed in [LRW+12]. *Lin et al.* have also developed REWARDS [LZX10], which instruments user space processes only using Intel Pin, captures the timestamps of each memory access and reveals their data structures in system-wide memory dumps. Another signature-based work is "Multi-Aspect, Robust, and Memory Exclusive Guest OS Fingerprinting" [GFP+14]. In this work *Yufei et al.* use invariants, derived from kernel code and data structures extracted from a memory dump of a system running Linux, to generate signatures to identify the kernel version. SigGraph [LRZ+11], instead, reconstructs the graph of data structures inside a memory dump of a generic OS, by using signatures derived from its source code. An approach based on multiple snapshots of the memory of the same process to collect information about the data structures used by the application is explored by *Urbina et al.* [UGCL14] and by *Feng et. al* [FPYL14]. (DeepMem [SYLS18]), instead, generate abstract representations for kernel objects by using a graph-based deep learning approach which requires, however, multiple dumps of the same OS in order to be trained, undermining its extensibility to unknown OSs. Other works have focused instead on the reconstruction of the shape, content and primitive types of the data structures [MCJ17, SSB10, SSB11, TDC10].

Researchers have also investigated how to create signatures from binary executables: ORIGEN [FPW+16] reconstructs offsets of atomic data structures by using static analysis on the code of a previous version of the same kernel, while *Case et al.* [CMR10] reconstruct the offsets of important Linux structures by directly analyzing code embedded in the dump itself. A different approach based on logic inferences on the position of the structure fields permits to *Qi et al.* [QQY22] to derive the offset of important fields in Linux kernel data structures.

However, in the case the fields' locations are randomized at compile time they cannot retrieve them. Two recent independent papers [PB21, FHA+22], instead, have developed a technique, based on a combination of symbolic analysis and code emulation, which is able to recover the structure offsets also in the case of their randomization. They can extract a Volatility profile from a Linux kernel memory dump without any other information, permitting the analysis also in the case when it is impossible to generate it at runtime.

All these techniques work on data structures that contain virtual addresses, thus assuming the problem of address spaces translation was already solved by other means, or they require multiple dumps of the same version of the operating system to train an algorithm. Furthermore, they assume that the analyst knows the OS and something about its internals (access to the source code, definitions of data structure to recover, multiple dumps of the same system in different conditions etc.), use approximations tailored for the OS for which they are designed or require to run the OS inside a hypervisor affecting their applicability in an OS-agnostic forensic analysis on real devices.

Chapter 3

From a Running Machine to the Physical Memory Dump

This chapter is extracted from "Is The End of Memory Acquisition As We Know It?", currently under submission to ACM Transactions on Privacy and Security (TOPS) for peer-review

3.1 Introduction

Memory forensics is a powerful technique that can provide valuable insights into the behavior and state of a computer system. It is routinely used to investigate and respond to computer incidents, but also as part of threat hunting, malware analysis, and intrusion detection procedures. The ultimate goal of memory analysis is to accurately recover the semantic meaning of data stored in the volatile memory of systems. Consequently, most research efforts in this area have been dedicated to overcoming the *semantic gap* and reconstructing an accurate representation of the content of the memory [DGSTG09, LRW+12, LZX10, GFP+14, LRZ+11, UGCL14, FPYL14, SYLS18, MCJ17, SSB10, SSB11, TDC10].

Several approaches can be adopted to acquire a copy of the physical memory during a memory forensics procedure. One option is to rely on specialized hardware devices that can access the content of the RAM memory over DMA. Another popular option is to use software-based memory acquisition tools that run within the target system. In this case, the analyst installs specialized software or leverages an integrated functionality of the operating system to copy the contents of the system's RAM onto an external storage device or transmit it over the network. Over time, multiple solutions [RFP+12, LHKF21, Ben23, avm23] have been developed to accomplish this task. In most cases, they rely on dedicated kernel modules that acquire the memory without altering the kernel behavior or interrupting its execution.

Independently of the fact that the memory is acquired by a software module or a hardware device, the acquisition process usually retrieves the content of physical pages in ascending order, according to their address in the physical address space. This technique is easy to implement and minimizes the logic required to track which pages have already been acquired, thus reducing the memory footprint of the acquisition tool. For this reason, this is also the technique adopted by almost all currently existing dump software for major operating systems such as Windows and Linux and it is the only available technique that can be used when the OS is unknown.

In fact, in the context of the OS-agnostic memory forensics [ODB23], the lack of information about the structure of the operating system prevents the analyst from giving precedence to those pages that contain information of potential interest for a forensic analysis [PFB19] or that can be more easily altered.

As a result, the fact that the content of a memory dump faithfully reflects the content of the system's memory at the time of its acquisition is a common misconception. In fact, since the memory is acquired while the system is running, memory dumps are not truly atomic snapshots of the system state but rather a collection of memory pages that are captured at different moments in time. This can lead to inconsistencies in the analyzed data and errors in the results of the investigation. For instance, previous studies have reported inconsistencies in page tables in at least 20% of the acquired images [CRI17] and a corrupted image once every five acquisitions [Ber18]. This problem was studied from a theoretical perspective by *Vomel and Freiling* [VF12] who introduced a set of definitions to describe the various requirements that a memory image must meet in order to be considered a faithful copy of the system's memory at a given point in time. The same authors have also tested various acquisition software and confirmed the presence of inconsistencies in the dumps they produced [GF16, VS13]. More recently, *Pagani et al.* [PFB19] have studied how to assess the atomicity of data structures commonly used by Volatility to help the analyst during the forensics analysis. However, despite the fact that these types of inconsistencies have been discussed in numerous papers, their presence and assessment has always been anecdotal and to date there are no studies that tried to precisely **quantify** the actual number of inconsistencies in a memory dump and their impact on the results of a forensics analysis.

3.1.1 Contribution

In this work, we employ the PANDA record-replay infrastructure to track all write operations performed by the OS kernel during a memory acquisition process. This allows us, for the first time, to precisely quantify the number and type of inconsistencies observed in non-atomic memory dumps and to pinpoint them to the exact field of the affected kernel data structure.

By gathering comprehensive data over a number of different experiments, we study how different acquisition techniques, target file systems, and operating systems influence the frequency of kernel writes. We proceed to classify and quantify potential types of inconsistencies that can impact page tables (page smearing), and evaluate the resulting implications for the accurate reconstruction of the virtual address spaces of the kernel and user-space processes. We finally track all the main kernel data structures that are used by Volatility [Wal17] to extract forensic artifacts from memory images, to determine the extent to which inconsistencies can undermine the reliability and validity of the results presented to the analyst.

The results are disconcerting. What the memory forensic community believed to be a sporadic inconvenience is instead a systemic problem that affects all memory dumps. Our experiments show that the occasional errors that are reported by analysis in their investigations are only the tip of the iceberg of a huge amount of inconsistencies that exist between a large fraction of pointers and kernel data structures. One of the pillar of digital forensics is that the analysis should be based on a pristine and faithful copy of the data. Our research shows that non-atomic acquisition methods violate this principle, not just by introducing small differences but by providing a completely untrustworthy picture of the state of the memory.

3.2 Problem Statement

A memory acquisition is considered *atomic* if the memory content remains unchanged from the beginning to the end of the acquisition process. This ensures that the result is indistinguishable from a hypothetical snapshot taken in a single instantaneous operation. From practical purposes, as pointed out by *Vomel and Freiling* in 2012 [VF12], the definition can be relaxed to a snapshot of the memory which "*does not show any signs of concurrent system activity*". This concept of atomicity allows for the collection of memory pages at different points in time, as long as the acquisition procedure maintains the causal relationships between memory operations and inter-process synchronization primitives. Although this definition is remarkably elegant, it poses significant challenges when it comes to practical measurement as it is very difficult to verify in practice. A more practical definition was introduced by *Pagani et al.* in 2019 [PFB19]. The authors said that a collection of physical pages, subject to some causal relations, is considered *time-consistent* if it exists a hypothetical atomic acquisition process that could have yielded the same outcome. In other words, there was a specific moment during the acquisition process when the content of those pages coexisted in the system's memory. If we consider the set of all kernel structures and their relations as a graph, a time-consistent dump produces a graph locally equivalent to one obtained from an atomic dump. However, at a global scale, the two graphs differ because the individual links between kernel structures retain their causality but the pages containing the structures are not acquired all at the same time. In order to produce a time-consistent dump the imaging tool must be aware of the relationships that exist between the various pages of the physical memory and must be able to find a dump sequence that satisfies all of them. While this is possible in theory, it is not feasible in practice. Specifically, when tools dump pages in a sequential order (which is the case with most available tools for major operating systems) achieving time-consistent atomicity is impossible if the entire OS is not frozen during the acquisition process.

In a running system, there are two main causes of inconsistencies: the virtual-to-physical address translation and the internal and concurrent activity of the kernel and user space programs. In modern kernels that support virtual memory abstraction, each program operates within its own distinct and private address space known as the virtual address space. The virtual addresses, used by a particular process, are translated into physical memory locations where the data resides through the combined efforts of the Memory Management Unit (MMU) and the operating system. On Intel and ARM architectures, this translation process requires in-memory data structures, hierarchically organized, prepared by the kernel and used by the MMU: the page tables [OB22]. Like all information contained in memory, page tables can also be affected by non-atomic acquisitions, a phenomenon called *page table smearing* [CRI17]. Page table smearing occurs when an acquired page table references page tables of lower levels whose content is modified by the kernel before they get acquired, resulting in inconsistencies and errors in the virtual-to-physical address translation which is later performed by the forensic analysis tool. As a result, the post-mortem analysis might miss entire regions of the virtual space, it can show inconsistent permissions bits (such as write permission, execution permission, or accessibility in the kernel or user space), or it can make errors in the reconstruction of the virtual memory of a process (e.g., by including data pages that originally belonged to other processes). This phenomenon can have serious repercussions on the results of the forensic analysis. Page table smearing can also occur on page tables related to the kernel memory or related to areas shared among multiple processes (e.g. for pages that host dynamically linked

libraries) further exacerbating the problem.

The second source of inconsistencies is the concurrent activity of the kernel and user processes during the memory acquisition process. The scheduling routine allocates the available CPU time among processes ensuring that each process is executed without monopolizing the system's resources. The kernel itself and all the other privileged software are subjected to these rules. So, when the analyst runs an acquisition tool, regardless of its level of privileges, its code runs along with other processes of the system[1]. This makes it so that the acquisition software cannot, in general, pauses all other processes, including the kernel, to prevent them from modifying the memory content during the dump procedure. This can introduce numerous inconsistencies within a memory dump because data structures allocated in distant regions in the physical address space are dumped at different times due to the scheduling policy. Things are even worse in multicore systems, in which multiple processes can run simultaneously on different physical CPUs further increasing the chances of inconsistency in the dump.

As a result, the kernel, with its ability to write to any physical page in the system and the ability to interrupt any process, is thus the major culprit in generating inconsistencies within a memory dump. Moreover, since memory forensic analysis of a system always requires to analyze kernel data structures, in this work we will focus only on inconsistencies in the address space of the kernel itself.

3.3 Measurement Technique

In our work, we want to quantify the inconsistencies that can be introduced by the kernel and the dump tool in the content and links among different *structures*. Structures are blocks of bytes that store data and have causal relations with other structures in memory. To clarify, they can be imagined as C `struct`s containing data and pointers to other structures of the same or different type. It is important to note that this definition includes both data structures used by the kernel linked by pointers containing virtual addresses, but also page tables used by the operating system and the MMU to translate virtual addresses into physical ones. In the latter case, the structures are the page tables themselves, and their entries can be seen as pointers, in physical address space, to other lower-level tables.

We consider the acquisition of a single 4KiB page as an atomic event because all the acquisition tools treat pages as atomic units and use a single call to the kernel to copy their content. In fact, kernel APIs within the low-level memory management system allows for mapping, allocating, and freeing only entire pages. This fact derives from the mechanism employed by the MMUs of modern CPUs to translate virtual addresses into physical ones: virtual addresses are treated as a combination of a page number, subject to translation, and an offset within that page.

To collect information about possible inconsistencies, it is essential to observe and monitor the kernel's activity throughout the entire dumping process. In our approach, we use PANDA [DGHH+15], an open-source tool, based on the QEMU whole system emulator, which offers extensive access to code and data within the guest environment. One distinguishing feature of PANDA is its ability to record and replay executions of an entire virtual machine, fa-

[1] If the CPU architecture supports more privileged modes than kernel mode, such as VMM and SMM modes in Intel CPUs, the execution of a dump tool in those modes cannot be interrupted by the kernel or user space programs, resulting in an atomic view of the memory used by lower privileged applications.

cilitating deep and interative system analyses. During the replay of an execution trace, PANDA allows, through its plugin framework, to perform sophisticated analysis and hook the system in many different ways (e.g., by triggering a callback when the CPU executes instructions at certain addresses, or by intercepting single writes on memory and reads the content of CPU registers).

In our experiments, we record the execution of an operating system (OS) in different conditions, for example during a memory acquisition process. Then, we replay the execution traces and analyze them with a set of custom plugins we designed to collect statistics about the activity of the kernel, the acquisition tool, and the state of data structures that we want to track. In particular, to monitor the evolution of the relations among structures during memory acquisition, we implemented a technique based on a versioning system. Before starting the replay we analyze the virtual machine memory to identify all the structures we want to track and explore their relations. For each tracked structure that appears in memory, we save the timestamp of its allocation and deallocation and we assign it a unique identifier. During the replay of the VM execution, whenever there is a modification to an interesting field or a pointer within a structure, we record the operation and, if necessary, explore the newly pointed structures. In addition, for each pointer that we track, we also maintain the unique index of the structure it points to. We use a unique index instead of its address because the kernel can allocate and deallocate different objects at the same memory location. When a page is dumped, we freeze the state of all the structures it contains: for each structure, we save the value of its fields and the value of the fields of the pointed structures. In this way, at the end of the replay, we can detect inconsistencies and also discriminate among their types, by checking whether the content of a pointed structure, saved when the page that contained it was acquired, is equal to its content saved when the pointing structure was acquired.

To study inconsistencies in memory dumps we follow a bottom-up approach: starting from individual pointers and moving up to the composite kernel data structures that are used in forensics analysis. We start, in Section 3.4, measuring the activity of the kernels of three different OSs in idle state. Then, in Section 3.5, simulating a process of dump, we collect statistics, for the same three OSs, about the relations among physical pages containing kernel pointers and how their physical page allocation strategy influences the number of inconsistencies in the memory image. In Section 3.6 we study how the dump tool, the different acquisition modes, and the target filesystem influence the content of the memory snapshot on Linux systems. Then, in Section 3.7 we classify the different types of inconsistencies involving kernel structures that can appear in a memory dump. Finally, we study the inconsistencies introduced by page table smearing in Linux and their impact (Section 3.8) and we measure the inconsistencies in Linux kernel data structures used by Volatility to extract forensics artifacts (Section 3.9).

3.4 Idle Kernel Activity

Before analyzing how, and how much, the activity of the kernel and dump tool introduce inconsistencies during the memory acquisition we collect some statistics about the interaction of the kernel with the memory in idle state (i.e., when left without user interaction nor external network requests for a sufficiently long interval of time).

For this experiment, we have chosen three OSs: Ubuntu and Windows 10, two of the most used and widely adopted OSs respectively in desktop and server environments, and vxWorks, a

Table 3.1: Statistics about kernels in idle state per minute.

OS	Writes on kernel address space (Millions)	Written Data (MiB)	Writes operations per size				Unique physical pages	Unique virtual address
			1-byte	2-bytes	4-bytes	8-bytes		
Linux	124	860	2.11%	0.36%	9.98%	87.5%	12,943	6,062,305
vxWorks	29	20	1.12%	0.71%	19.14%	78.90%	15	2521
Windows 10	389	2538	6.62%	3.55%	11.90%	77.93%	8878	6,724,883

popular real-time OS used in industrial devices and the Internet of Things, but rarely discussed in memory forensics literature. We recorded, using PANDA, the execution trace of an x86 VM equipped with 4GB of RAM, running the three different 64-bit OSs.

We booted each machine and left it without any interaction for 30 minutes before performing the experiment, to minimize the activity of the OS. We then recorded each kernel for 10 minutes, based on the estimated time required by existing tools to acquire 4GB of memory (as we will describe in more detail in Section 3.6).

Table 3.1 summarizes the results. Among the three operating systems, Windows 10 exhibits the highest activity, with approximately 3 times more writing events per minute than Linux, and approximately 3 times more data written in terms of quantity. As expected, vxWorks is instead the less active OS, performing only one-thirteenth of the memory operation compared to Windows, probably due to its real-time setting which tends to minimize kernel activity in favor of tasks. Each of the three operating systems demonstrates a notable preference for 8-byte writes (average 81.5%), possibly due to its optimized use for copying large data quantities. On the other hand, 2-byte writes appear to play a minor role in their operations, accounting for only 1.55%. This suggests that kernel data structures' fields and variables of this size are relatively uncommon. It is also interesting to observe that while Linux writes less often to memory, it touches the highest number of unique physical pages (1.45 times more than Windows). However, due to the higher number of write events per minute, on average, Windows performs more writes per page per minute, approximately 4.5 times more than Linux.

For the Linux kernel, we can classify write events further, as the kernel's virtual address space is divided into regions of fixed sizes [ker23]. The majority of writes events (62.03%) happen in the `vmalloc` region which contains not-physically continuous pages used to allocate buffers that require to be contiguous in the virtual address space. `vmalloc` region is used, in particular, to allocate memory for video framebuffers and this could explain the high number of writes and at the same time the low number of unique virtual addresses written (only 0.48%). 33.28% of write operations are performed on the direct mapping region which permits the kernel to write directly on every physical page of the system. This region contains the majority of the unique kernel virtual addresses written during the execution (97.35%). Another 1.67% of the operations are performed on the virtual memory map area containing the `struct page` data structures used by the kernel to track the physical pages used in the system. And finally, in the remaining 3.02% of the events the kernel writes on its internal global variables and modules, and in per-CPU pages.

Table 3.2: Statistics about kernel pointers inconsistencies.

OS	Pointer inconsistencies (absolute)	Pointer inconsistencies (relative)	Average distance among pointers (MiB)
Linux	1,182,366	14.6%	1120
vxWorks	12,733	11.1%	13
Windows 10	1,009,818	26.6%	1739

3.5 OS Memory Allocation Strategies

The most common approach to obtain a copy of the system memory is to acquire each physical page in ascending order, based on its address in the physical address space. This linear dumping strategy is adopted by most software acquisition tools for all major operating systems like Windows and Linux.

To study how different OSs use physical pages, how they reference physical pages and the impact of the linear acquisition technique on the kernel pointers we have replayed the execution traces of the same VMs used in experiments of Section 3.4 running a PANDA plugin that emulates the linear acquisition of the memory. The plugin, by starting from the first physical page available, at a constant rate, saves a hash of the content of each physical page. At the same time, it scans the page by looking for 8-byte values that can be valid kernel virtual addresses. The plugin validates candidate addresses by checking if they are correctly resolved to a physical address by the MMU and if they are part of the canonical range associated with the kernel memory (that for the three OSs in our dataset starts at `0xffff800000000000`). These addresses represent possible pointers to kernel data structures referenced by some structure contained in the page under dump. Therefore, by saving the hash of the pointed physical page and by comparing it with the one computed when the page is subsequently acquired, we can identify the presence of pointer inconsistency.

At the start of the dump, our PANDA plugin saves the content of the page tables of the OS to collect information about how the different OSs manage and maps the physical pages onto the kernel virtual address space. Linux and vxWorks, for example, map the entire physical address space by using 2MiB large physical pages, in order to directly access each physical page available. This allows them to reduce the number of page table entries needed to address each physical page. When they need to allocate a smaller amount of memory, they generally create a new, separated, mapping by using 4KiB physical pages (sometimes removing the previous 2MiB one). VxWorks tends to create these mappings by using adjacent regions of the physical space, Linux uses more fragmented but still compact regions, while Windows 10 uses large pages to map only a fraction of the entire physical address space and selects random 4KiB pages across the entire physical address space to map smaller regions. On Windows 47% of the physical pages that contain a kernel pointer are mapped using only 4KiB mappings while, on Linux, this value is only 19% (vxWorks has kernel pointers only in 2MiB pages).

Table 3.2 shows the number and ratio of inconsistent pointers in the three OSes. In absolute terms, both Windows and Linux contain over 1 million each, while from a relative point of view, we can see that Windows encounters this problem more frequently (likely due to the much higher frequency at which he updates pointers in memory), resulting in potential inconsistencies in over

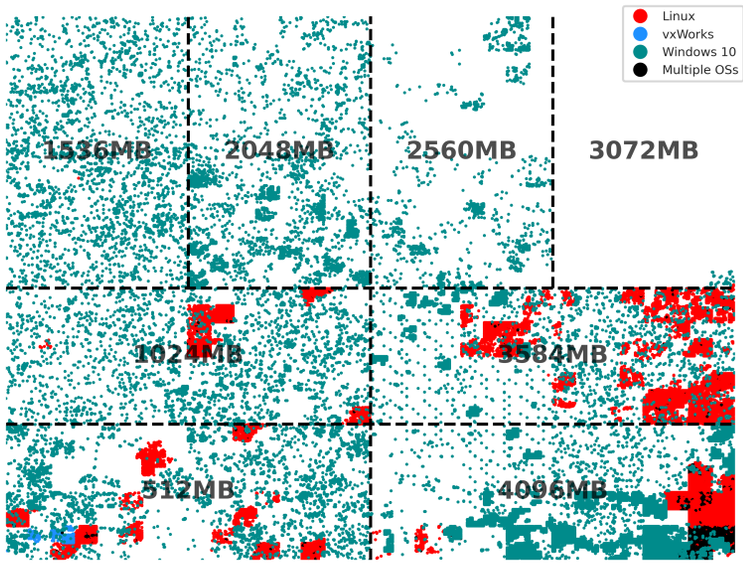


Figure 3.1: Pages with at least one pointer inconsistency

25% of pointers in kernel memory. The last column shows the average distance (in the physical address space) between a pointer and its pointed data. On 4GB of memory, this distance is 1.1GB for Linux pointers and an astonishing 1.8GB for Windows pointers. For vxWorks, where the memory is all allocated in contiguous chunks, the average distance is instead only 13MB. This is confirmed by Figure 3.1, which shows a Hilbert plot of the actual pages that contain one or more inconsistent kernel pointers. Linux inconsistencies happen in chunks, Windows ones instead are spread all over the physical memory, while vvWorks ones are all located in a small area on the bottom-left of the figure.

As a reminder, the values provided in this section should be considered as an upper bound on the number of pointer inconsistencies in the acquired memory, as our measurement cannot have false negatives but false positives are possible due to the way we identify candidate pointers (i.e., a large integer value that falls within a valid memory range would be considered as a potential pointer).

A more precise measurement would require a detailed knowledge of the OS internals and data structures, which we will present in the following Sections.

3.6 Impact of the Acquisition Technique

So far we measured the activity, and possible inconsistencies, of a kernel in an idle state. However, during the acquisition process, the acquisition tool itself introduces additional noise in the system, as it requires to read and write a copy of the entire memory to a storage or network device. To measure this impact, we restrict our focus to the Linux kernel, as it is easier to explore its internals and to modify acquisition tools to collect additional information required by our tests.

User- vs Kernel-based Acquisition

The first factor that affects the acquisition noise is whether the acquisition is performed completely in kernel mode or from user space by taking advantage of an already-existent mechanism to access to the entire memory of the system. Kernel-based acquisition is much more common, and it is used for instance by the LiME tool [Ben23]. LiME is a Linux kernel module that allows to dump the entire physical memory of a Linux system by copying it to a non-volatile storage or by sending it through the network. After it is loaded, the tool traverses the `iomem_resources` linked list of the kernel to identify the regions of RAM. It then proceeds to dump them by processing one page at a time in ascending order of physical page addresses. Instead, Microsoft AVML [avm23] is a dumping tool that uses the virtual device `/dev/kcore` to access the system RAM from the user space. As LiME, AVML dumps the physical pages in ascending order permitting to save them on a file but managing the entire dump process from the user mode.

We expect user-space solutions to trigger more activity (and therefore potentially more inconsistencies) in the kernel. In fact, when running these tools the kernel has to copy the requested page to user space. The tool can then perform some operation (e.g. adding a hash or compressing the data) and then sends the result back to the kernel to write it onto the disk. These additional back-and-forth copy force the kernel to perform more operations, overwrites more pages, and therefore compromise even more the atomicity and efficiency of the acquisition process. To confirm this hypothesis we recorded the execution trace of a VM Ubuntu 22.04 desktop machine (Kernel 5.19.0-40) equipped with 4GB of RAM while executing either a LiME or AVML acquisition.

During the execution of LiME the kernel performed 5576 million write operations, affecting 573.861 different physical pages and a total of 38 GiB of data. The execution of AVML caused instead 9710 million write operations on 796.709 different physical pages, for a total of 68 GiB of data (1,78 more than Lime), As expected, acquiring the memory from user-space almost double the amount of kernel write operations. As such, this approach should be limited as much as possible, preferring those in kernel space whenever available. Since our purpose is to quantify the inconsistencies that occur in a dump under the best possible experimental conditions, we will use LiME as a memory acquisition tool for the rest of our experiments.

Storage

We now look at the impact of the target storage that is used to record the acquired memory image. In particular, this covers the interplay of different aspects, including where the image is saved (network vs internal disk vs USB disk), how (by using kernel functions or direct access to the device), and which filesystem is used to store it.

LiME has three main operating modes that can alter the content of the memory during the dump process:

- **Dump on disk** In this mode, LiME uses kernel primitives to write the content of the memory on a file. The specific type of target hard drive, an internal SATA drive or an external USB drive, can impact the number of physical pages marked as "dirty" by the kernel during the dumping process. This is because the kernel copies the content of a page being dumped into various caches, including bus and device drivers, ring buffers, page cache and others. Furthermore, different target filesystems use, internally, different algorithms, caches and data organization resulting in different memory footprints generated

during the writing operations.

- **Dump on disk using Direct I/O** In this mode, LiME opens the target file using the `O_DIRECT` flag permitting to bypass kernel's page cache reducing the number of physical pages dirty. However, direct I/O requires to be supported by the underlying filesystem.
- **Dump through the network** In this mode, LiME sends the content of the RAM through a TCP connection established by an external system.

To measure the effect of the different LiME dump modes on the system we use a minimal VM running Linux 6.2.10 equipped with 1 GiB of RAM. To minimize the impact of kernel operations on memory due to user space syscalls, the system runs only two userspace processes: `getty` and `sh`. After waiting for the kernel to enter the idle state, we dump the memory on an emulated USB external hard drive and on an emulated internal SATA drive. We test 8 commonly used file systems, in buffered and direct I/O mode, as well as dump the memory using the network. LiME has been loaded with the options `timeout = 0`, to disable the default timeout beyond which LiME ignores the page and `format = raw`, which instructs LiME to copy the memory as is, without adds LiME file format headers[2].

Table 3.3 presents the measurement results conducted on file systems using an external USB disk, an internal SATA disk, and a network dump. In each column, the highlighted value is the best result among the test cases of the same type. The most significant indicator in order to determine how much the dump method dirties the memory content is the number of different physical pages written during the dump. As we can see in Column 5, for dump performed using buffered I/O there are no significant differences, for the same file system, between the number of physical pages written on a USB disk or SATA one (a maximum difference of 0.23%) and, as well as, there are no significant differences in the number of pages written among different file systems (a maximum difference of 1%). However, different file systems can have different performances in terms of time needed to complete the dump and the number of writing events. In particular, on a USB disk, XFS requires 45% less time and 56% less writing operations than the widely adopted FAT32 file system. These differences can have an impact on the number of inconsistencies inside a memory dump because more time and operations are required to complete the dump process the higher is the probability for the kernel to write on pointers located on pages already dumped.

Only 3 file systems in our tests (Btrfs, exFAT and FAT32) support direct I/O in kernel mode. In this case, as we can see from Column 3, the number of writes on MMIO regions is increased by a factor of 100, a sign that the kernel has changed the method to access the physical device to directly write on them. In all three file systems, we also observe a drastic reduction, up to 99.8%, in the number of different physical pages written during the dump process. This very positive aspect is however counterbalanced by a dramatic increase in the time required to acquire the memory, which increased by up to 100 times. To understand the reasons for this huge increase in dump time we have analyzed calls to the allocation/free functions used internally by the kernel to manage disk write operations. During our investigation, we found that for every page written on disk, the Block I/O subsystem of the kernel allocates and releases a `struct bio` along with its associated `mempool`. The memory for these operations is obtained from the `bio-160` memory pool, which, in turn, relies on the kernel SLAB memory

[2]This last option is required also to enable direct I/O, because the LiME file format headers introduce data unalignments incompatible with `O_DIRECT` file option.

Table 3.3: Statistics about the dump process in different conditions.

Mode	Writes on kernel address space (Millions)		Writes on MMIO regions		Total size (GiB)		Unique physical pages		Time required (ratio)	
	USB	SATA	USB	SATA	USB	SATA	USB	SATA	USB	SATA
Btrfs	874	811	59824	37778	6.01	5.58	249340	249204	1.81x	1.53x
exFAT	1005	938	96112	61772	6.89	6.43	251074	250728	1.79x	1.33x
Ext4	818	757	60692	35696	5.61	5.20	249421	248873	1.76x	1.16x
Ext4 no journal	776	719	61744	36443	5.33	4.95	249439	248864	1.78x	1.10x
F2FS	951	910	61329	36743	6.51	6.23	249406	249379	2.04x	1.33x
NTFS	796	739	61329	38711	5.48	5.09	249411	249129	1.75x	1.31x
FAT32	1404	1317	84456	89542	9.65	9.06	250328	250908	2.39x	1.82x
XFS	632	569	57605	34255	4.41	3.97	249405	249041	1.62x	1x
Btrfs D. I/O	49137	37708	9147061	6707022	344.04	265.19	75037	78885	255.76x	73.00x
exFAT D. I/O	10698	4713	5204034	3950081	73.20	32.11	466	497	109.34x	15.85x
FAT32 D. I/O	16657	6000	9138277	5773820	114.15	40.88	1125	1127	236.71x	19.58x
Network	1336		1000453		8.64		488		2.73x	

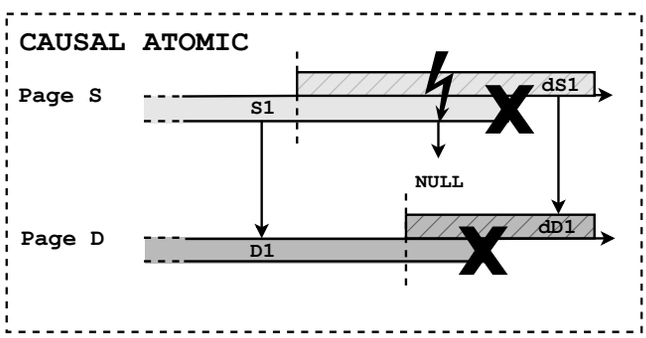


Figure 3.2: Example of causal atomic dump.

pool system. All of these memory operations significantly contribute to an increase in the dump time. Furthermore, unlike buffered I/O, which optimizes disk writes by using the page cache and adapts to the underlying hardware, direct I/O mode can reveal differences due to the distinct interfaces involved. We have observed a significant disparity in the time required to perform a dump on a USB drive compared to a SATA drive. This discrepancy is not limited to PANDA artifacts, as we have also observed it on real hardware. It can be attributed to the different implementations of the USB and SATA kernel subsystems, as well as the varying complexity and number of layers in their respective protocol stacks.

Finally, the network dump is the most balanced method in terms of physical pages modified (only 22 more than the fastest direct I/O method), amount of time needed (1.5x the fastest buffered I/O file system), total number of events, and total size of data written, suggesting that this should be the preferred acquisition method, whenever available.

3.7 Types of Inconsistencies

So far we have discussed possible inconsistencies in relation to kernel pointers that connect two physical pages. However, in the case of structures of known sizes, such as page tables and kernel

data structures, we can perform a more in-depth analysis and classification of different types of inconsistencies. In our measurement, we consider five different types of inconsistencies that affect memory structures. Four of them are related to the order and time difference between the acquisition of the pages that contain two different causally-related structures. The last is instead caused by a timing difference in the dump of two or more pages that contain fragments of a single large structure.

In order to visualize the various inconsistencies, we will use a series of diagrams, similar to the one shown in Figure 3.2. The diagram shows two structures `S1` and `D1` allocated in physical pages `S` and `D`. Both structures have a size that is less than or equal to a physical page size, which we consider as an atomic dump unit within this study. The horizontal axis of the diagram represents time. Structure `D1` is referenced by structure `S1` through a pointer, depicted by a vertical arrow on the left side of the figure.

At a specific moment, indicated by a vertical dashed line, the dump tool saves the contents of page `S`, and consequently, the contents of structure `S1`, to the dump file. From this point on, in Figure 3.2, there are two bands associated with the structure `S1`: the lower band, which continues to represent the structure `S1` in memory, and the upper knurled band `dS1`, which represents the unmodifiable copy of structure `S1` saved within the dump file. After a while, also the page `D` is dumped, resulting in a split of the associated band.

After the dump of page `D` the kernel performs a memory write operation, represented as a thunderbolt in Figure 3.2, invalidating the reference to structure `D1`. Finally, the kernel deallocates both structures, an operation represented as a cross in the diagram.

As shown in the right part of the figure `dS1`, the dumped copy of `S1` structure correctly contains a reference to `dD1`, the dumped copy of `D1` structure: the dump of the two structures and their relation is causal atomic since the causal relationship between the two structures is preserved by the dump.

We can classify inconsistencies into two categories: causal inconsistencies (the upper schemas in Figure 3.3) and value inconsistencies (the bottom ones). In causal inconsistencies, the causal relation between structure `S1` and structure `D1` is compromised and the structure `dS1` in dump file does not point to the correct structure `dD1` but to generic data that can be located at the same address. In value inconsistencies, instead, the causal relation among structures is preserved but the content of the pointed structure `D1` is changed during the dump process.

Here we explain in detail the 5 types of inconsistencies (the first four of which are represented in the diagram in Figure 3.3):

- **Type 1 (causal inconsistency)** The kernel de-allocate the destination structure `D1` after dumping `S1` but before dumping `D1`, potentially replacing the memory with different data. As a result, the dumped `dS1` structure references unrelated data and the dump does not contain the original content of structure `D1`.

- **Type 2 (causal inconsistency)** The kernel allocates a couple of structures with the referenced one in an already dumped page. It is the mirror case of Type 1 when the destination structure `D1` is dumped before `S1`.

- **Type 3 (value inconsistencies)** The kernel modifies the pointed structure before it is acquired but after the dump of `S1` structure: `dS1` in the dump will reference the modified version of `D1` instead of the original one. It is important to note that there is a second possible case that produces this type of inconsistency. This situation can arise when, after

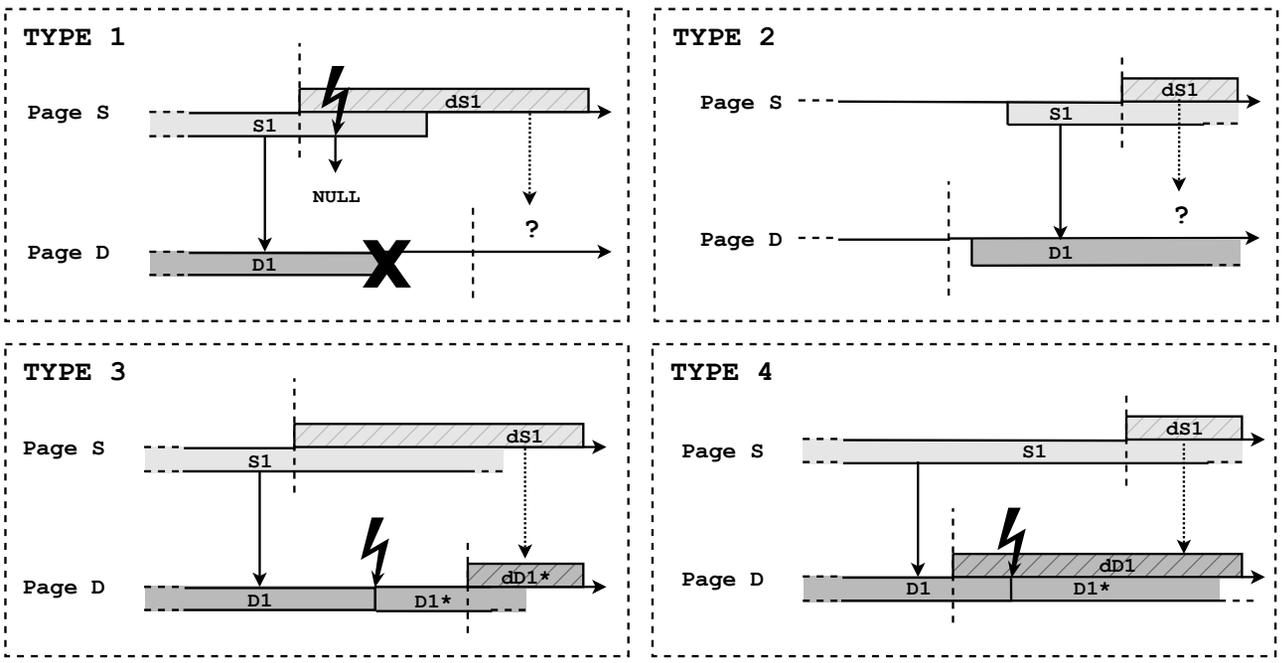


Figure 3.3: Types of inconsistencies.

discarding structure `S1`, the kernel frees up structure `D1` and substitutes it with another structure of the same kind that has no causal connection to `S1`. In this scenario, within the dumped data, there exists a structure of the same type as `D1`, but it is impossible for the analyst to determine whether it represents a modified version of `D1` or a subsequent allocation. An example of such a case can occur in Linux if the `D1` structure is a part of a kernel SLAB.

- **Type 4 (value inconsistencies)** It is the mirror case of Type 3 when the destination structure `D1` is dumped before `S1`.

- **Type 5 (value inconsistencies)** This inconsistency, not represented in Figure 3.3, occurs when a structure occupies more than one physical page. In this scenario, after the dump of a page that contains a portion of the structure, the kernel modifies one of the other pages composing it: the version of the structure saved on the dump then will consist of fragments obtained at different points in time. As a result, the values within the dumped structure will not be globally consistent with each other.

3.8 Page Table Smearing

The first type of memory structure that we study in our experiments are page tables, as page table smearing (i.e., the presence of inconsistent or malformed page tables in memory dumps) is often reported as one of the main problem of non-atomic acquisitions.

To study this phenomenon we developed a PANDA plugin that tracks the creation/destruc-

Table 3.4: Types of inconsistencies in page tables.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
Type 1	-	-	-	-	35	-	302	-	1	-
Type 2	1	-	-	3	22	6	45	-	63	-
Type 3	-	3	6	33	57	26	198	-	60	22
Type 4	2	3	5	21	24	40	24	10	23	21

Table 3.5: Processes with page table inconsistencies.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
Type 1	-	-	-	-	4	-	19	-	-	-
Type 2	1	-	-	-	3	1	2	-	3	-
Type 3	-	2	2	9	12	9	18	-	17	11
Type 4	2	2	3	6	6	10	4	3	9	11
Unique processes	2	2	4	11	14	12	26	3	24	19

Table 3.6: Dumps with at least a kernel page table with inconsistencies.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
Type 1										
Type 2				✓			✓			
Type 3		✓		✓	✓	✓	✓			
Type 4				✓	✓		✓		✓	

tion of radix trees in the system[3], allocations and deallocations of page tables of lower-levels[4], and all write events. We performed our analysis on an Ubuntu 22.04 desktop machine (Kernel 5.19.0-40) equipped with 4GB of RAM, and repeated each experiment ten times to reduce the effect of randomness.

For each run, we booted the machine and started a set of commonly used applications. After 30 minutes of inactivity the kernel and user applications consumed about 25% of the available memory. Then, we started a memory acquisition using LiME and dumping the memory on an emulated external USB drive formatted with the Ext4 file system. We decided on this configuration because an external disk allows the analyst to preserve the content of the hard disk of the target machine (often a requirement in an investigation) and because Ext4 (see Table 3.3) introduces a minimal overhead compared with the other filesystem (e.g., FAT32) whose support is compiled by default in the kernels of all major distributions.

The running applications are chosen to simulate an office environment and include: `firefox` with 6 tabs opened, `libreoffice`, the `thunderbird` E-Mail client, and the `evince` PDF reader.

[3]Intercepting calls to functions `pgd_ctor` and `pgd_free`.

[4]Intercepting calls to functions `___pte_free_tlb`, `___pmd_free_tlb`, `___pud_free_tlb`, `pud_free_pmd_page`, `__free_pages_ok` and `free_unref_page`.

Before the start of the acquisition, we counted an average of 210 processes running on the system. The LiME acquisition took 9.18 ± 5.46 minutes to complete (on 5 dumps it requires less than 5 minutes while in one case, D_4, up to 17 minutes). This variability in the acquisition time could derive from the activity of the user space processes which can slow down the dump requiring the kernel to perform privileged operations for them: LiME does not use any mechanism to prioritize its activity such as starting a high-priority kernel task to dump the memory from.

Table 3.4 reports, for each of the 10 dumps, the number of inconsistencies categorized by type. All dumps exhibit inconsistencies, with a minimum of three and a maximum of 569. Table 3.5 shows how many processes have an address space affected by the aforementioned anomalies and Table 3.6 counts the memory dumps with inconsistencies in the kernel page tables. Note that we separated the two, as otherwise all inconsistencies would be reflected in all processes as, for the Intel architecture, kernel page tables are present in the radix tree of all processes in the system. Overall, inconsistencies in the kernel page tables occurred in 60% of the acquired images, while the number of unique affected processes in each dump ranges from 2 to 26.

It is important to note that the level at which the inconsistency appears in the radix-tree determines how much of the virtual address space of a process is impacted: the closer it is to the root of the tree the greater its impact. Cumulatively, across the 10 dumps we have found 91 inconsistencies that impact entries of the top-level page tables (level 0, the nearest to the root), 25 of level 1, and 901 of level 2.

For inconsistencies of Type 2 and 3 we can distinguish two cases: when the affected page table is not deallocated and reallocated but only modified by the kernel during the acquisition, and when it is instead deallocated and substituted by another one.

In the first case, we can quantify exactly how much of the virtual address space is affected by the anomaly: on average 770 KiB (with a standard deviation of 2.85 MiB) and up to a maximum of 64.63 MiB of the virtual address space of affected user processes present anomalies such as missing pages, wrong permission or errors in translation. In this case, the analyst might miss pieces of evidence because he is not able to fully explore the memory of the kernel or of the user-space processes.

In the second case, things are even worse, as the page tables of one process can erroneously point to page tables of another one. As a result, the address space of the first process will contain and refer to data belonging to the second process without the analyst being able to tell the difference. This virtual address space anomaly has the potential to deceive analysts in many ways. For instance, it can create the illusion that pieces of code from one process exist within the address space of another, implying potential manipulation or injection of code. Moreover, it can lead to a scenario where sections of one process's heap are replaced by fragments from another, rendering the analysis of the process's data structure unattainable. As rare and unlikely as this may seem, in our dataset we have found two dumps, D_4 and D_6, in which this anomaly occurs. In particular, in D_6 two of the office applications that we have started, (`libreoffice` and `thunderbird`) have erroneously attributed memory pages that belong to the graphical environment manager (`gdm`).

To make things worse, while in theory (even though for the best of our knowledge no tools attempt to do so) one could try to detect, but not correct, causal inconsistencies (Type 1 and Type 2) the presence of a value inconsistency (Type 3 and 4) cannot be recognized nor corrected. In fact, the presence of a false page table introduced by a causal inconsistency could

Table 3.7: Types of inconsistencies in kernel structures.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
Type 1	22	37	6	16	5936	16	4733	7	131	7
Type 2	10	-	2	135	1455	11	1546	-	2227	7
Type 3	1773	1740	670	3496	9772	3551	4040	570	1400	597
Type 4	1961	2010	1272	5315	4245	4733	3183	1225	1981	1321
Type 5	-	-	-	-	-	-	-	-	-	-

Table 3.8: Average number of inconsistencies per kernel structure.

Struct Type	**Struct Size (bytes)**	**Number of Instances**	**Inconsistencies**				**Percentage with inconsistencies**
			Type 1	Type 2	Type 3	Type 4	
`cred`	176	2248	-	-	-	-	-
`dentry`	192	322,248	687	2630	13,445	16,686	7.36%
`dentry_operations`	128	118	-	-	-	-	-
`fdtable`	1188	1108	-	6	34	127	15.07%
`fdtable array`[1]	-	1107	21	67	59	148	19.42%
`file`	232	80,302	243	151	218	207	1.02%
`file_system_type`	72	402	-	-	-	-	-
`files_struct`	704	1108	-	-	9	-	0.81%
`fs_struct`	56	1134	-	2	61	-	5.56%
`inode`	632	253,983	-	40	-	-	0.02%
`mm_struct`	1048	1104	10	17	14	10	4.08%
`sock`	760	7302	-	-	-	-	-
`socket_alloc`	768	7210	35	52	19	41	2.04%
`super_block`	1472	236	-	1	-	10	4.66%
`task_struct`	9792	2132	62	29	312	412	22.56%
`vfsmount`	32	488	-	2	8	18	5.74%
`vm_area_struct`	208	348,551	9853	2396	13,430	9587	9.19%

[1] This is a variable size array of pointers to `struct files` pointed by `fdtable.fd` field.

be detected by employing a validation model that checks the page tables against the inviolable constraints defined by the MMU, as recently discussed in Chapter 4. However, in cases of value inconsistency, the erroneous page tables introduced may still appear valid as they have correct layouts accepted by the MMU. The automatic recognition and correction of such anomaly can be quite challenging, and even a human analyst may encounter difficulties in identifying it as an inconsistency.

3.9 Kernel Structures

So far we have seen that memory acquired in a non-atomic way contains a large number of inconsistencies among pointers and page table entries. In our last set of experiments we now look at those inconsistencies that affect fields of kernel data structures that are actually used

by existing tools to recover evidence from a memory dump.

In particular, we consider as forensics-relevant data structures, all those that are used by at least one of the base set of Volatility plugins. Likely, some of them are rarely modified during a normal execution. For instance, the `modules` linked list (composed by `struct module` nodes) maintains information about kernel modules loaded in the system and therefore only experiences modifications when a module is loaded or unloaded from the system, an operation the analyst should avoid while acquiring a snapshot of the system memory). Therefore, the occurrence of non-atomic inconsistencies among these structures is exceptionally rare in practice. For this reason, we will focus our analysis on those structures that are routinely modified by a running kernel.

In particular, among those traversed and explored by Volatility's plugins, we selected 17 important kernel data structures that are frequently modified by the OS. A complete list is shown in Column 1 of Table 3.8. These structures are responsible for maintaining highly volatile forensics-relevant information about the running processes, their resources, opened files and sockets, and part the file system information cached in memory.

To quantify the presence of inconsistencies in these structures, we created another PANDA plugin that tracks the evolution of the content and links among kernel data structures during an acquisition process. Before the start of the dump, the plugin explores the content of the memory of the VM identifying kernel data structures and dereferencing pointers contained in them. Then, during the replay of the execution trace, it intercepts allocations of new data structures and deallocations and modifications of already tracked ones, by maintaining information about the version of each structure as already explained in Section 3.3.

For each of the 17 different structure types, we track separately two types of fields: the pointer fields that maintain relations among data structures and that are traversed by Volatility plugins to reach other structures, and the data fields that contain the information retrieved by the plugins. In total, in our experiments we tracked 38 pointers and 40 data fields. It is important to note that this number corresponds to the subset of fields used by Volatility and not to all the fields present in the data structures, since many of them have no forensics relevance.

Table 3.7 reports the number of data structures that were affected by at least one inconsistency in each of the ten images. As in the case of page smearing, each dump contained at least one value inconsistency (Type 3 and 4) and a causal inconsistency of Type 1, in total ranging from 1950 to over 20K inconsistencies.

Again, as in the case of page smearing, dumps D_4 and D_6 had the highest number of structures with inconsistencies. At a closer look, these two dumps contain the largest total number of tracked structures allocated and deallocated by the kernel during the dump process, with 275,830 and 327,907 respectively (the average is $185,151$). This suggests that the kernel was more 'active' during the acquisition process. This additional kernel activity can be attributed to various factors that occur during the normal operation of the system, even in an idle state. For example, memory allocation and opening/closing of file handles by system daemons, the launch of scheduled processes like automatic system update managers, or internal kernel optimizations, such as flushing page caches.

In our kernel configuration, there are 9593 different types of kernel data structures[5] and only 103 of them (1.07%) have a size greater than a physical page. Among those that we

[5]It is possible to count them accessing to `BTF` information stored in the kernel using `pahole` utility.

track for this experiment, only the `task_struct` belongs to this category (with a size of 9792 bytes). This makes Type 5 inconsistencies, which happen when the kernel modifies a large structure that spans across more than one physical page of memory, extremely rare events among forensically-relevant structures. In fact, as shown in the last row of the table, we did not encounter any of them in our experiments.

To better understand how the inconsistencies affect the different types of kernel data structures in Table 3.8 we report the size of each structure, the mean value of the number of structures present in a dump, the mean absolute value of the number of data structures affected by inconsistencies and the ratio between the mean number of unique structures affected and the mean number of structures of that type. The four more common structures (`vm_area_struct`, `dentry`, `inode`, `file`) account for over 98% of the structures we track in memory and contain 98% of the inconsistencies. Unfortunately, these are also the most used by Volatility plugins, as we discuss in more detail later in this section. For example, inconsistencies in data structure related to cached information about physical and virtual filesystems such as `dentry` and `inode` can compromise the ability to extract files from volatile filesystems like `/tmp`. Often, these files are not saved on disk but reside solely in RAM, making their recovery challenging when data structures are compromised. It is also important to note that while there are relatively few instances of `task_struct`, they contain the highest number of inconsistencies among the traced data structures (22%). In particular, they are affected by a large number of value inconsistencies: this means that dereferencing a traced pointer in a `task_struct`, the pointed structure has, roughly, a 1 in 4 chance to contain information not in sync with those of the `task_struct` itself. This can mislead analysts in diverse ways. For instance, if the inconsistency impacts the `cred` pointer, a process may falsely exhibit lower privileges than it truly possessed. Similarly, if the inconsistency affects the `files` pointer, the analyst might encounter difficulties in identifying all the file descriptors opened by the process.

A similar amount of inconsistencies are also present in the variable-sized arrays that hold pointers to the `file` structures that hold the file and socket descriptors open by a process (19%) and the `fdtable` structures that connect the `task_struct` to the variable-sized arrays (15%).

In the case of Types 1 and 2 inconsistencies, it is possible that a pointer that in memory refers to a type of structure, in the dump file points instead to a different type of structure. In our dataset, this occurs in three different dumps: D_4, D_5, and D_6. D_6 in particular contains 279 of these cases, mainly related to the `vm_file` pointer in the `vm_area_struct` which reference the `file` structure associated with a memory-mapped file (167 times), the `vm_next` pointer (30 times), or the `rb_left` and `rb_right` pointers of the substructure `vm_rb` (45 times).

These two last cases are closely related to each other from the point of view of a forensics analyst: `vm_next` pointers maintain the linked list of virtual memory areas of a process and are used by various Volatility plugins to explore process memory. If an analyst suspects that a `vm_next` pointer is corrupted, or tampered, she can use a different Volatility plugin, `linux_proc_maps_rb`, to explore the virtual memory areas of a process: this plugin walks the red-black tree which links together, along with the linked list, all the `vm_area_struct`. However, our experiments show that 18 `vm_area_struct` in D_6 present value inconsistencies in the linked list *and* in at least one field of the red-black tree, while in 2 cases in both fields. In this case, the analyst cannot recover the areas of the virtual address space in any way, as all paths used by Volatility are corrupted. This situation can lead to several issues. For example, the `linux_library_list` module, designed to report all libraries loaded by a process, relies on exploring the `vm_area_struct` linked list to identify these libraries. If this linked list is

corrupted as well as the red and black tree, it is possible that a library injected by a malicious process into another might not be identified.

Table 3.9: Fields involved in inconsistencies per plugin.

Plugin	Fields causal inconsistencies	Fields value inconsistencies
linux_check_creds	1	-
linux_check_inline_kernel	9	3
linux_check_syscalls	2	3
linux_dump_map	1	2
linux_elfs	9	1
linux_enumerate_file	2	3
linux_find_file	2	3
linux_getcwd	5	-
linux_info_regs	1	1
linux_ldrmodules	3	2
linux_library_list	3	1
linux_librarydump	3	2
linux_list_raw	6	-
linux_lsof	8	1
linux_malfind	3	3
linux_memmap	1	-
linux_mount	2	-
linux_netstat	2	1
linux_plthook	3	-
linux_proc_maps	12	6
linux_proc_maps_rb	1	1
linux_procdump	1	1
linux_process_hollow	3	2
linux_process_info	9	4
linux_psaux	1	1
linux_psenv	1	-
linux_pslist	1	2
linux_psscan	-	-
linux_pstree	2	1
linux_recover_fs	2	5
linux_threads	3	3
linux_tmpfs	2	1
linux_truecrypt	3	3

Finally, Table 3.9 reports for each plugin considered in our study, the number of tracked fields that contain inconsistencies in our dumps. These can affect the output of Volatility plugins in terms of capability to extract information (causal inconsistencies) and data reliability (value inconsistencies). As the table shows, all plugins are affected by at least one inconsistency with the only exception of `linux_psscan`, which in fact relies on carving techniques to extract the `task_structs` from a dump. As a result, plugins can report incorrect information for all dumps with the exception of D_0 and D_1. This is due to the fact that almost all the basic Volatility

plugins need to traverse one of the top 4 data structures per number of inconsistencies. In these cases, as suggested by Pagani et al. in [PB19], it would be beneficial to use alternative paths to traverse a 'more stable' kernel data structure. By relying on multiple paths to reach the same information, the analyst might be able to identify and overcome inconsistencies of Type 1 and 2. However, inconsistencies of Type 3 and 4 would still not be detectable automatically since paths through these types of inconsistencies would produce wrong but plausible results that would be hard to identify in an automated fashion.

3.10 Impact and Discussion

Both hardware-based and software-based memory acquisition, with the exception of research prototypes that rely on a hypervisor or otherwise higher privilege level, do not stop the kernel during the acquisition process.

The fact that this can lead to inconsistent and/or corrupted information is known since at least 2005, when the initial prototypes of memory forensics tools were first proposed [**?**]. However, there has been only anecdotal information reporting 'unusable' memory images, without any data to understand how often these errors are encountered in practice. For instance, while researchers have previously noticed errors in the page tables for roughly 20% of the memory images they acquired [CRI17], we show that the problem is much more widespread and that **every single** memory dump we acquired contained at least one (but more frequently many) of those errors.

In fact, our results show that memory acquired even from an idle operating system can contain inconsistent pieces of information. While the vast majority of them are likely irrelevant for a forensic investigation, some are not. For instance, every dump we acquired contained at least a few processes with errors that affected their page tables – in some cases resulting in fragments of the memory of one process to be attributed by mistake to another one. Even if we restrict our analysis to only a few dozen data structures used by Volatility to extract information from a memory dump, we find tens of thousands of actual inconsistencies. For some structures, almost one out of four instances contained inconsistencies in the field used by Volatility. Whether these errors can undermine an investigation or lead to wrong conclusions depends on the questions the analyst is trying to answer. In most cases, they might not be a problem, but the fact that forensic analysis is based on such a fragile foundation is disconcerting.

Moreover, our numbers are in reality very conservative and represent just a lower bound of the inconsistencies one might expect to encounter in real investigations. This is due to the fact that PANDA does not support the emulation of a machine with more than one CPU processor/core. On modern systems with more than one core/socket, inconsistencies can only increase because of the larger amount of writes per second performed by the kernel, and because of the synchronization mechanisms (such as semaphores and locks, which are additional variables in the kernel memory) required to manage the interaction of multiple CPU units. Moreover, we focused on the best case scenario in which the memory of the machine was only 25% occupied and the machine was left untouched and without network activity both during the acquisition process and for 30 minutes prior to that. In real settings, the memory usage might be higher (thus leading to more fragmentation, more data structures, and a larger number of page tables) and/or the computer might need to maintain its operation during the acquisition process (for instance when memory is acquired from servers as part of threat hunting or incident response

investigations). Finally, even the size of the memory can have an impact, as a larger amount of RAM (even if unused) means that physical pages might be further apart and therefore increase the distance in time between their acquisition. Because of all these factors and conditions, we expect our results to only show the *best* an analyst might expect during an investigation, while the number of errors due to the non-atomic acquisition will most likely increase considerably in real-world settings.

Even more worrying is the fact that in most cases, the analyst is unable to tell whether the answers he gets are correct or not. Small imprecision might occur very often without even being noticed. Because of this, we believe that the way we perform memory acquisition today would need to be revised.

Possible ways to mitigate the problem exist. For instance, in 2019 *Pagani et al.* [PFB19] proposed an acquisition tool that tried to acquire causally-related data structures one after another, instead of blindly collect physical pages in sequential order. The tool is not supported anymore and does not run on recent Linux kernel, but we believe that that is an interesting direction that could help significantly reduce the number of inconsistencies. Our experiments also show that the choice of the acquisition method and target filesystem can have a large impact on the amount of data structures modified by the kernel and on the acquisition time; this is another factor that should be taken into account before acquiring the memory of a system.

Finally, we hope research in this area continues and will lead in the future to the development of new solutions to perform atomic memory acquisition.

3.11 Related Works

Over time, numerous studies [Kor07, LK08, HBN09, MC13] have pointed out how the lack of atomicity in memory dumps can lead to problems and errors during a forensic analysis. The first to formalize the concept of atomic dumps were, in 2010, *Vomel and Freiling* [VF12] that have also introduced the distinction between integrity correctness and atomicity of a dump. In a later work, *Gruhn and Freiling* [GF16] have evaluated 12 different acquisition tools that adhered to these three criteria. In 2021 *Freiling et al.* have extended the concept of atomicity and integrity also to disk snapshot in [FG21]. More recently, Case and Richard highlighted the pressing issue of page smearing [CRI17]. With the prevalence of servers equipped with large amounts of RAM causing longer acquisition times, the smearing effect has become increasingly common. In 2019, *Pagani et al.* [PFB19] have introduced "temporal dimension" in memory forensics. This dimension provides the analyst with an initial means to evaluate the atomicity of the data structures employed during an analysis. Additionally, a series of experiments were conducted to demonstrate that page smearing not only introduces inconsistencies in page tables but also indiscriminately impacts any analysis involving structures located in user space.

Among the first to propose a solution to the problem of non-atomicity of memory dumps, even before the formalization of the definition, were *Huebner et al.* who in their work [HBHW07] propose an automatic acquiring of the state of the kernel and user applications periodically performed by the operating system itself. This solution, however, requires a redesign of the internals of the kernel which is not immediately applicable to existing OSs.

An interesting approach to obtaining an atomic snapshot through software acquisition is developed by *Schatza and Bradley* [Sch07]. At runtime, they inject a minimal kernel that stops

the execution of the running OS and dumps its memory. This approach, as underlined by the authors, have several limitations due to the strong integration between the dumping kernel and the original OS and the variability of the hardware that must be supported.

In 2009, an approach based on the data remanence effect in memory chips after the reboot of the system is used by Forenscope [HSH+09] to acquire the memory in an atomic way. This technique, known as cold boot memory acquisition has been proven, unfortunately, that strongly depends on the chipset and memory modules used, with certain combinations of components failing to retain RAM content upon reboot [CBS11].

The next year *Martignoni et al.* have introduced HyperSleuth [MFPC10] which uses a custom hypervisor injected at runtime to perform memory dumps using dump-on-write and dump-on-idle in order to be resistant to evasive malware, that can tamper the OS kernel, and obtain an atomic dump at the same time. This promising technology, however, requires activation of virtualization support on the CPU prior to booting the machine itself, and also is not applicable in cases where the operating system itself acts as a hypervisor such as Windows 10 virtual secure mode.

Recently, in *"Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots"* [FHA+22] the authors have collaterally introduced a Linux kernel module that allows atomic memory dumps. However, the developed technique, by temporarily blocking system interrupts, can cause multiple, chain-like crashes of the kernel modules responsible for hardware management at the end of the dump process.

Other solutions developed to obtain atomic dumps involve using modified firmware to run the dump tool at higher privilege levels (as SMM module [RFP+12]) or with the operating system no longer running but with the memory contents still available (as an EFI module [LHKF21]). These two methods, however, require deep modification to the system firmware residing on the motherboard or to be installed in the EFI partition and a valid cryptographic signature to permit to the dump tool to run in SecureBoot environment which is now a default condition for modern operative systems on Intel CPUs.

Chapter 4

From the Physical Memory Dump to Virtual Address Spaces

This chapter is extracted from "In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics", published in an article appeared in the ACM Transactions on Privacy and Security (TOPS) Volume 22, Issue 4, July 2022 [OB22]

4.1 Introduction

The problem of recovering semantic information from low-level data is common to many areas of computer security. In particular, this is the main obstacle when performing a physical memory analysis—a task that is key for both memory forensics and virtual machine introspection. The problem, often called the *semantic gap*, captures the challenge of "interpreting low level bits and bytes into a high level semantic state of an in-guest operating system" [FL12]. However, at a closer look, the semantic gap can be further divided into two different aspects: the reconstruction of the virtual address spaces (which deal with translating pointers expressed as virtual addresses to their physical position in the memory) and the recovery and identification of key operating system (OS) kernel data structures (e.g. those related to running processes, memory management, and kernel modules).

In practice, most tools and existing techniques neglect the first aspect. For instance, Libvmi, a popular virtual machine introspection library, explicitly mentions that the virtual-to-physical translation is only available on live VMs as it would otherwise require OS-specific heuristics [XLXJ12]. Similarly, *Fu and Lin* [FL12] acknowledge that to analyze the content of the memory, the first step requires "to perform the MMU level virtual to physical (V2P) address translation" but the authors again avoid the problem by inspecting the registers of a live target. Other papers focusing on narrowing the semantic gap, such as Virtuoso [DGLZ+11], relied instead on pre-existing frameworks (e.g., Volatility [Wal17] and Rekall [Coh14]) for the virtual to physical translation. These memory forensics frameworks, as well as products developed by BlackBag [bla23] and Volexity [Vol23], rely, in turn, on a set of manually curated and very specific OS and architecture heuristics to locate kernel data structures that are used to recover the mapping between the virtual and physical address spaces. However, these heuristics are based on a deep knowledge of the internals of the OS under analysis, thus precluding their generalization to different operating systems or different CPU architectures. In other words,

every memory analysis study to date has "avoided" the virtual memory translation step either by focusing only on live systems or by considering only a small subset of OSs (in practice, Linux, Windows and OSX) running only on the x86 architecture and, partially, on ARM (in particular Android and recent Apple M1 devices). However, as recently shown by *Cozzi et al.* [CGFB18], IoT malware authors started to target also architectures less common in desktop environments, such as PowerPC and MIPS. This is troublesome from both an academic and a practical aspect. In the academic community the virtual-to-physical address translation, which is a fundamental step of the semantic gap reconstruction, is considered a solved problem. However, except for a few OSs running on x86 and ARM, this is far from being true. At the same time, from a practical point of view, the lack of cross-architecture OS-independent analysis techniques to perform virtual to physical translation poses a serious challenge for the future of memory analysis. In fact, the rapid increase in IoT devices and cloud-hosted VMs translates into a more variegated number of architectures and operating systems.

4.1.1 Contribution

In this work, for the first time, we systematically study how to bridge the semantic gap in virtual-to-physical translations in 10 different CPU architectures in an OS-agnostic way. We show how the traditional page tables, which most researchers are familiar with, is only one of many possible ways to perform the virtual to physical translation and how the memory management unit (MMU) affect the reconstruction of virtual to physical mappings. In each case, we study the inviolable constraints imposed by the MMU and use them to build signatures to recover the required data structures from memory without *any* knowledge about the running OS. We also introduce a static code analysis step to recover the state of the MMU registers configured by the OS at boot time. We implement these techniques in a tool that, in contrast to the existing forensics tools available, uses only parameters derived from the CPU ISA to recover virtual to physical mappings. We test our tool against memory dumps collected from 26 different operating systems and a physical device, proving the usefulness and accuracy of the described techniques in the reconstruction of the virtual to physical mapping in real-world scenarios. Finally, we show a possible practical usage of our technique to recover and analyze the user space processes running on an unknown device. This permits an analyst to start a forensics analysis on the system without any knowledge of the internals of the OS.

4.2 Virtual Memory: Basic Concepts

The term *Virtual Memory* refers to an abstraction of the memory resources available on a given machine. In virtual memory mode, each program performs memory accesses using an isolated and private address space called virtual address space (VAS). The combined work of the MMU and the OS permits to translate virtual addresses of a particular process into physical addresses that specify where the data is physically located in memory.

The virtual memory abstraction provides numerous benefits, allowing programs to be written as if they had complete and unrestricted access to the entire physical memory, regardless of its usage and the presence of other programs. This permits to run multiple processes at the same time without worrying about the actual physical memory configuration, which can change across different types of machines. Virtual memory provides also isolation and protection be-

tween processes by preventing malicious, unauthorized, or poorly implemented programs to access the memory of other running programs. Furthermore, it allows a program to allocate memory beyond the limits of the physical address space (PAS) by mapping part of the virtual address space of a process onto secondary storage. like a hard drive, thus allowing the process to allocate more memory than the physical RAM installed on the system.

The virtual memory abstraction is implemented in different architectures by relying on two different techniques: *segmentation* and *paging*, as illustrated in Figure 4.1. Segmentation, the oldest of the two methods, was originally developed to organize and protect running programs. Its goal is to divide the virtual address space of a process into one or more logical units, called *segments*, with different sizes and access permissions, by mapping them into another address space called segmented address space (SAS). In general, segments follow the internal structure of the program that they represent: the memory of a process can be divided, for example, into two different segments containing, respectively, the code and the data. Segmentation, however, does not permit optimal use of the available memory: if we map segments of various programs directly on the physical memory, the physical address space starts to fragment and at some point it is impossible to allocate new chunks of sufficient size to accommodate new segments.

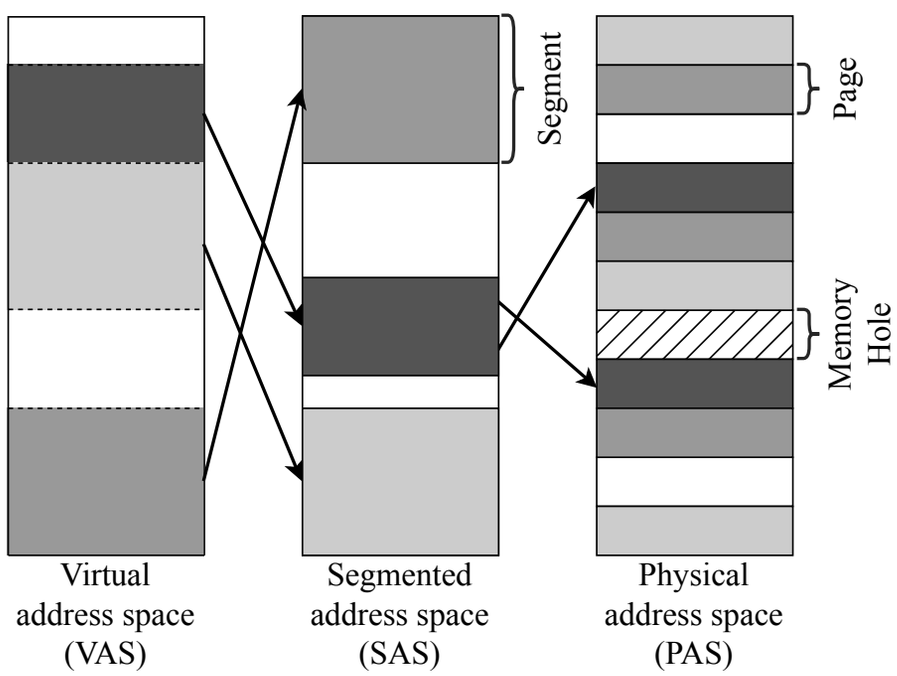


Figure 4.1: Virtual, segmented and physical address spaces. Memory holes are physical memory regions that are not usable as storage memory regions (e.g MMIO, device reserved regions, ROMs, not assigned regions etc.)

To solve this problem we first need to introduce the concept of *page*, which is a contiguous block of memory of fixed (and typically small) size. Then we divide the segmented address space into a set of pages (called virtual pages), and we define a way to map them to pages in the physical address space (called physical pages) as shown in Figure 4.1. This technique, called paging, drastically reduces the fragmentation of the physical memory and contributes to maximizing the use of the available resources. In the context of our study, it is important

to understand that both segmentation and paging require some in-memory data structures, or dedicated CPU registers, that need to be configured by the OS, and used by the MMU.

4.2.1 MMU

The MMU is the hardware device that converts the virtual addresses used by the processor to physical addresses. To accomplish this task, the MMU needs to be configured by using special system registers, while in-memory structures that maintain the virtual-to-physical mapping have to be defined and continuously updated by the operating system. When the MMU fails to resolve a requested virtual address, it raises an interrupt to signal the OS to update the in-memory related structures.

It is important to note that the MMU demands strict conformity of the shape and topology of the in-memory structure to the ISA and MMU configuration requirements. Otherwise, it raises an exception and aborts the address translation.

The translation process can involve up to two separate translations, both accomplished by the MMU: segmentation, which converts virtual to segmented addresses, and paging, which converts segmented addresses to physical ones. Some architectures use either one or the other, while others use both.

In general, when the system boots, the MMU is virtually disabled and all the virtual addresses are identically transformed into physical ones. This allows the OS to start in a simplified memory environment and gives it time to properly configure and enable the MMU before spawning other processes.

Since address translation is a performance bottleneck, the latest translated addresses are cached in a few low-latency hardware structures called Translation Lookaside Buffers (TLBs). Before starting a translation, the MMU checks if TLBs contain an already resolved virtual address and, if so, it returns directly the corresponding physical addresses.

In our analysis of the most common CPU architectures shows that two main approaches are used for paging: radix trees, and inverted page tables.

4.2.2 Radix Tree

Radix trees maintain a hierarchical representation of the segmented address space. Each tree is composed (see Figure 4.2) by N-1 levels of *directory tables*, each containing entries that either point to tables of the lower level or to a physical memory page (*huge pages*), whose size depends on the level itself. The final level is composed of *page tables* that point only to same-size physical memory pages. The tree root is the physical address of the directory table of Level 0 and it identifies uniquely the segmented address space and, consequently, the process to which it is assigned.[1] This address is stored in a special system register (here generically called `RADIX_ROOT`) by the operating system and it is used by the MMU to locate the radix tree when it starts a new translation. The translation performed by the MMU starts from the root table pointed by the address contained in `RADIX_ROOT`: the MMU then divides the segmented address into two parts: a prefix and a page offset. The prefix part is divided into a series of N chunks that represent the hierarchical sequence of indexes to be used to locate the entry inside a table of the corresponding level. This process ends when an entry points to a physical

[1] In certain architectures, the kernel does not have its own radix tree but is instead mapped inside every process segmented space at a fixed continuous block of addresses.

page. At this point, the MMU returns the concatenation of the page offset extracted by the segmented address to the physical page address found in the last page table entry.

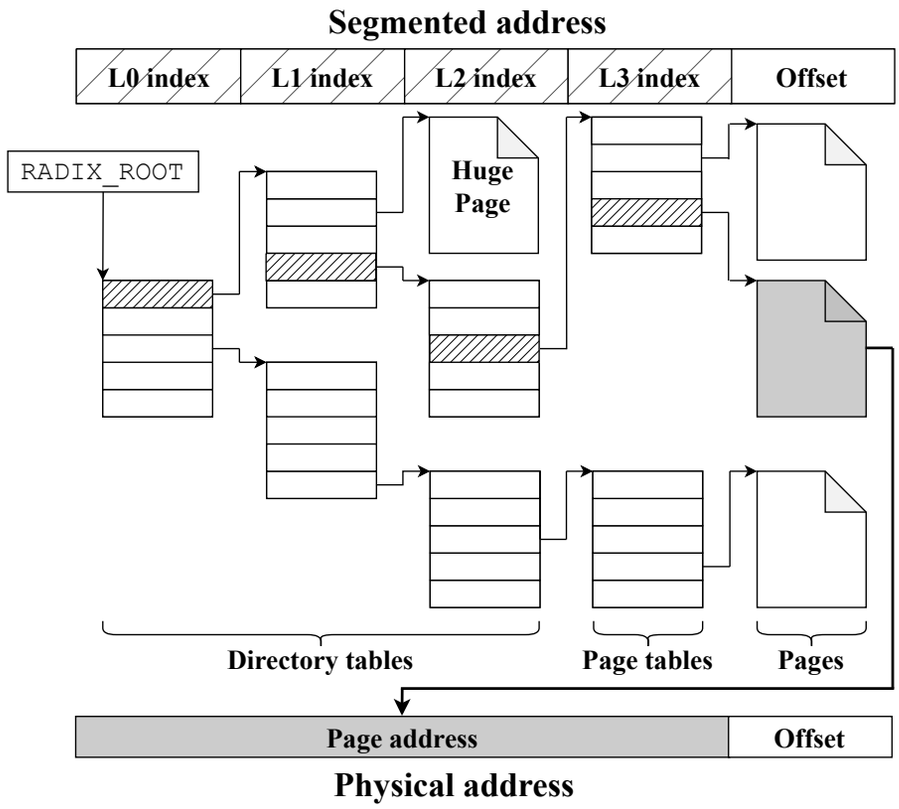


Figure 4.2: Radix tree address resolution process.

4.2.3 Inverted Page Tables

Inverted page tables maintain one entry for every physical page in the system along with the associated virtual page address in its segmented space. As such, instead of having one radix tree per segmented space, a single inverted page table is maintained for all the processes of the system. Inverted page tables are often implemented by using a *hash table*, whose address is stored in a special system register. The OS then needs to use the same hash function used by the MMU, as defined in the processor ISA, to index an entry in the hash table when allocating memory for a process. When the system needs to resolve an address, (see the right side of Figure 4.4) the segmented address is split into the page number and a page offset. Starting from the page number and a unique segment identifier, the MMU generates a hash value which is retrieved from the hash table and, in the case of a positive match, the physical address is formed by concatenating the physical page address, indexed by the hash, with the page offset extracted from the segmented address.

4.3 Approach

The goal of our work is the automatically recover the virtual address space (virtual address space) of the kernel and all user space processes contained in a memory dump, independently from the OS and the applications that are running on the machine. The only input to our analysis is the CPU architecture and a copy of the raw memory. In case the user is not even aware about the hardware in which the memory was collected from, the first piece of information could also be automatically recovered from the memory itself, for instance by using tools like `cpu_rec` [cpu23]. However, since this is a separate and orthogonal problem, for simplicity we assume that the analyst is already in possess of this information (which is anyway the case in any conceivable use case scenario). To reconstruct the set of virtual address spaces we need to extract the configuration of the MMU and also locate and interpret the data structures that are used by the MMU to translate virtual to physical addresses. While the actual techniques we employ are architecture-specific, and therefore we will present them in detail in the following sections, we can summarize them in three broad categories.

4.3.1 Structural Signatures

The use of structural signatures computed over the values of individual fields (sometimes called *invariants*) has been used in several studies to locate particular data structures in memory dumps or binary blobs. This technique has been successfully applied to retrieve OS kernel structures [DGSTG09, LRZ+11], application data [LRW+12], user space stack layouts [AD07], internal representations of malicious processes [CSXK08], and hypervisors/VMs configurations [GLB13]. We follow a similar approach by manually studying the different MMU data structures and compiling a list of structural constraints that can be used to build pattern-matching signatures.

For instance, a set of constraints to match Intel x86_64 ISA directory tables require each entry pointing to an inferior table to have the `P` bit set (`entry[P]==1`), some other bits unset (`entry[MAXPHYADDR:51]==0` $\wedge$ `entry[7]==0`) and the pointer containing a paging table to be in the physical address space range (`entry[addr]` $\ll$ `12` $\in$ `RAM`). These constraints are then translated to patterns that can be used to verify whether the bits of a given physical page satisfy the requirements. The complete list of constraints we use is reported in Appendix 6. It is important to stress that these signatures are derived solely by analyzing the inviolable constraints imposed by the MMU and they are completely independent of the OS.

4.3.2 Validation Rules

Structural signatures are used to match all the possible regions of memory that *could* contain a given data structure. However, the patterns are not very unique and therefore result in a large number of false positives. Therefore, the second type of technique we use is a set of rules to filter out the noise and reduce the number of candidates. However, since we cannot make assumptions on the underlying OS and we already use all the information from the MMU to distill the signatures in the first place, possible validation routines are scarce and difficult to construct.

The problem is further complicated by the great variability of OS behaviors in managing the system resources. For instance, one might think to discard tables that contain pointers to

physical pages outside the size of the physical memory itself. However, even this simple rule fails in practice as some OSs create a complete radix tree that can address all the possible physical memory pages of the RAM, whether such memory is installed or not on the machine. Furthermore, all OSs maintain a mapping to MMIO ranges that are often allocated far outside the physical address space. Another simple strategy would be to set a minimum threshold on the number of pages (but this fails as the kernel can map the process virtual address space using a single physical page) or to look at how the physical pages of a process are distributed in the physical memory and discard the structure if not even two pages are consecutive. However, even this simple idea did not work as we encountered operating systems that spread physical pages all over the entire physical address space and, in general, is false for all the OSs in the presence of high memory pressure.

To avoid these problems, our strategy to create validation rules is based on the use of other inviolable constraints imposed by other CPU subsystems. In particular, we aim at identifying pages that *must* always be mapped in the virtual address space to have a functional system (for instance, the ones containing the Interrupt Address Table) for certain architectures and use them to discard those virtual address spaces that do not map their physical addresses.

4.3.3 Binary Code Analysis

Some architectures use MMU-related CPU registers that contain values that do not refer to any in-memory data structure. Thus, to recover these values, we cannot rely on signatures. However, we can still take advantage of how the boot process works: when a machine boots up the bootloader loads the kernel code into physical memory and runs it. In this early boot stage, the MMU is disabled and all the virtual addresses are mapped in a fixed way to the physical ones. At this point, the kernel sets up the MMU-related registers based on the physical configuration of the machine (e.g. the CPU model or the size of the RAM). When we dump the physical memory of the system, this contains all the allocated kernel code[2] and this allows us to try to recover the content of the MMU registers by analyzing the assembly instructions used by the kernel code to load their values. In particular, for architectures that have aligned and fixed-size opcodes, we can search for the byte patterns corresponding to the instructions that load values inside the MMU-related registers. Then we can identify the functions containing these opcodes and finally, we can use data-flow analysis to derive which values are loaded inside them. To accomplish this task, default values of some CPU registers at the system power-on are required because the kernel could use them to decide how to configure the MMU.

In the next three sections, we discuss eleven of the most popular CPU architectures, grouped into three separate groups based on the different MMU modes available to them. The first group includes ARM, RISC-V and x86 in their 32 and 64-bit implementations. These architectures use exclusively radix tree-based MMU modes. The second group is composed of the ISA derived from the POWER architecture: PowerPC (32-bit) and Power ISA (64-bit). These two ISAs rely on an inverted page table as their main MMU mode but, at the same time, also possess other peculiar modes which distinguish them from other architectures. Finally, the last group contains MIPS in its 32 and 64-bit flavors. These architectures show unique and very flexible management of the virtual memory, which requires a separate category.

[2]It is possible that the MMU setup is done by the bootloader and the kernel can delete or rewrite the physical memory pages containing it. However, all the OS we encountered in our study leave the task of the configuration of the MMU to the kernel itself.

4.4 Group I: Radix Trees

This group includes three architectures that use a radix tree to translate virtual to physical addresses. We present them here ordered from the easiest to the hardest to analyze.

4.4.1 RISC-V (32 and 64-bit)

MMU Internals

RISC-V is an open-source ISA first published by UC Berkeley in 2010. It has become increasingly popular as several large companies [Bur20, Won20, Kre17] have announced its use in their future products, or they are implementing their custom versions. The last ratified ISA [WA19] features both 32-bit and 64-bit little-endian CPUs that support three different MMU modes: `SV32`, `SV39` and `SV48` (the last two available only for 64-bit CPUs). As there is no segmentation unit in RISC-V, virtual addresses are identically mapped to segmented ones. The MMU uses the `SATP` register as `RADIX_ROOT` register, which also permits to select the MMU mode. The radix tree is composed of two, three, or four table levels (respectively for `SV32`, `SV39` and `SV48`) with a predefined size and fixed layout of the entries specified by the ISA. Every mode supports both 4KiB pages and huge pages of different sizes.

Analysis

In RISC-V the shape of the entries permits to distinguish among tables of different levels, thus allowing the reconstruction of a radix tree in a very simple way. The algorithm starts by parsing the memory dump and identifying directory tables of all levels, page tables, and data pages by using the set of constraint signatures presented in Appendix 6. Starting from data pages and empty tables we then look for all tables which point to them and recursively list all tables of level N-1 pointing to the ones of level N in a forward search. By using this technique we can completely reconstruct the radix tree, find the top-level table (and its address contained in the `SATP` register), and derive the virtual address space associated with it. Furthermore, the privilege separation and access permission bits, present in the table entries, permit to distinguish user pages from kernel ones, and pages containing data from those containing executable code.

4.4.2 Intel x86 (32 and 64-bit)

MMU Internals

First introduced in 1985 by Intel, the x86 ISA has been extended over the years by adding new functionalities and new instructions, maintaining however full backward compatibility. This fact has resulted in the existence of a segmentation unit that must be enabled to use the paging one [Int20].

Before the existence of the paging unit, OSs running on x86 ISA used segmentation as a form of isolation between the processes running on the system. However, after the introduction of the paging unit, segmentation was quickly replaced by paging by all OSs. As we could not find any case in which the two were used in combination, for the 32-bit version of the architecture we assume that the operating system defines segments that identically map virtual addresses to segmented ones.

When AMD introduced the x86_64 architecture extension (also known as AMD64), aware of the fact that the segmentation unit was not used by any OSs, it virtually disabled it: when the MMU is configured for using the 64-bit specific paging mode it continues to operate with the segmentation unit enabled but it uses an automatically defined set of segments, which identically map all the virtual addresses to segmented ones.

The current implementation of the x86 architecture supports three major MMU modes with paging enabled: 32-bit mode, PAE mode, and 4-level mode, the last one available only in 64-bit. The physical address of the root of the radix tree is stored in the `CR3` register and the radix tree is composed, respectively, by two, three, or four table levels with a predefined size and fixed layout of the entries. Every mode supports 4KiB pages, as well as huge pages of different sizes.

Analysis

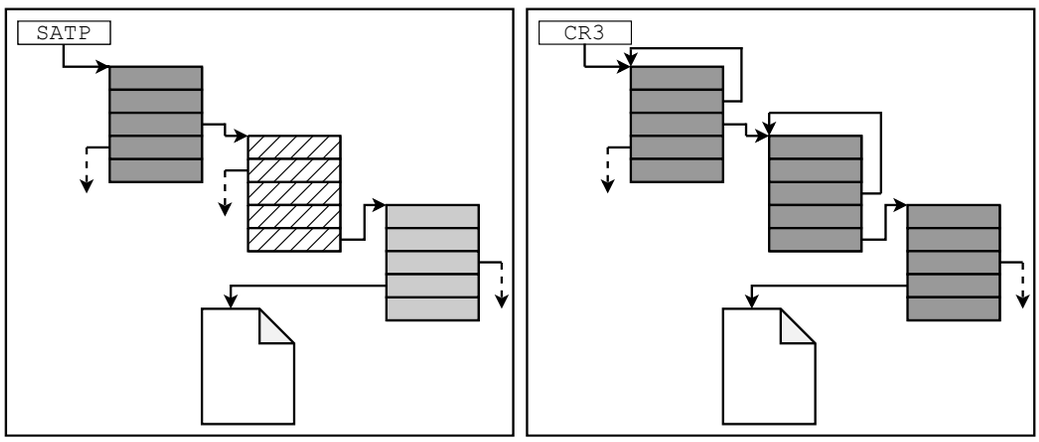
In contrast to the RISC-V ISA, the x86 directory table and page tables are, in general, not distinguishable. This peculiarity is exploited by some OSs (for example the Windows family) to allocate a reduced number of tables by inserting self-references entries in them (see Figure 4.3). This poses a problem for our analysis as it is difficult to tell whether a parsed table is the root of a radix tree, an intermediate level, or a leaf one.

To solve this problem we use another structure that must be initialized by any operating system and that has a predefined format: the interrupts table. In the x86 architecture, the interrupts descriptor table (IDT) has a predefined length and is composed of entries with a fixed format. Each entry contains the virtual address of an interrupt handler to which the CPU jumps when it receives an interrupt. Every process must have mapped the IDT in its virtual address space because an interrupt can occur at every moment also when the system is not running kernel code. This allows us to discriminate between root directory tables and lower-level tables: true top directory tables are starting points of radix trees that are able to resolve all the virtual interrupt handler addresses while trees with false-positive top directory tables, in general, are not able to do so. As for the RISC-V architecture, it is possible to infer which ranges of the virtual address space of a process contain code or data and its privilege level thanks to the flags available in the tables entries.

4.4.3 ARM (32-bit)

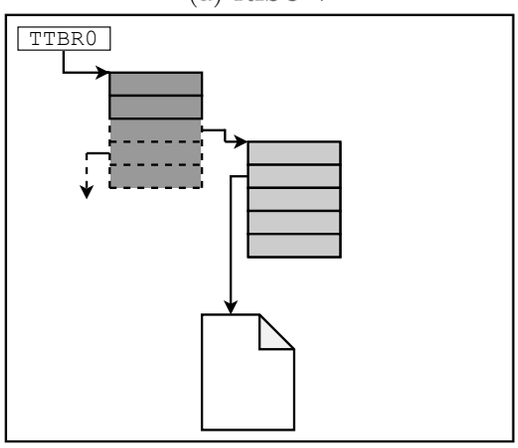
MMU Internals

ARM is a RISC (Reduced Instruction Set Computer) architecture developed by ARM Holdings and widely adopted in mobile and IoT devices. ARM-based processors are produced in different implementations and versions by many companies. In our study, we consider only the Application Profile ARM processors (ARM-A) which is the only one implementing an MMU. ARM32 CPUs are bi-endian and support two MMU modes [Hol18]: short and long descriptor modes for which the radix tree has, respectively, two and three levels and maps 4KiB pages as well as huge pages of different sizes. Each virtual address space is divided into two parts. The high-end one is resolved by using a radix tree pointed by the `TTBR1` register, contains the kernel code/data, and is accessible only in that mode. The lower end is instead resolved by using the `TTBR0` register, which points to a different radix tree, containing the data structures of the

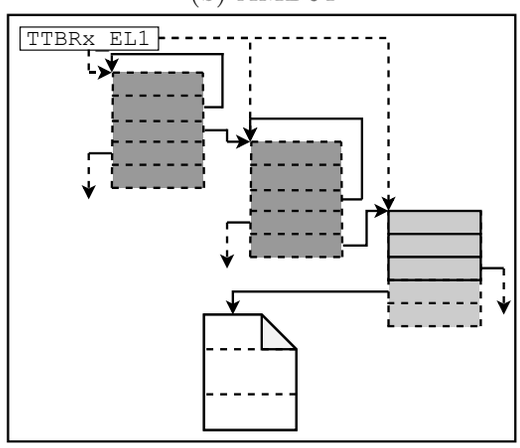


(a) RISC-V

(b) AMD64



(c) ARM32 (user process)



(d) AArch64

Figure 4.3: Radix trees used by RISC-V, AMD64, ARM32, and AArch64. Tables with different layouts are represented with different colors. Tables and pages with partially dashed borders can have different sizes. Tables with full dashed borders shall not exist in the tree. Note the possibility for x86 and AArch64 tables to have self-referencing entries.

running process.[3] At every change of the execution context, the scheduler loads the value of `TTBR0` register accordingly to the scheduled process while the value of `TTBR1` is left unchanged. The `TTBCR` register controls which MMU mode is used and the size of the kernel and user virtual address space, from which it derives the size of the user space top-level tables in short descriptor mode.

Analysis

Because of the peculiarities of the ARM architecture, an approach based solely on signatures (like the ones used for RISC-V and Intel) is not sufficient. In fact, we first need to recover the value of the `TTBCR` register to identify the virtual address space layout. For this reason, ARM32 requires to first perform a code analysis phase described in Section 4.3. If the procedure succeeds, we then continue our analysis based on the retrieved information.

[3]It is also possible for the OS to use only `TTBR0` as `RADIX_ROOT` register for both the kernel and the user parts of the virtual address space. This configuration is used, for example, by Linux.

As it was for RISC-V, also in ARM short descriptor mode the tables can be easily classified as page or directory tables. We can then parse the memory looking for kernel directory tables and page tables, which have both a fixed size, and user process directory tables of sizes compatible with the recovered value of `TTBCR`.

If the MMU is configured in long descriptor mode, things get more complicated. In this mode, it is not possible to distinguish between directory tables of different levels but only between directory and page tables. Moreover, in ARM we cannot rely on the IDT to filter out false radix trees because its entries do not have a fixed format, which makes them difficult to locate in memory. To solve the problem, we look for pages containing code that must always be present to have a fully functional paginated system and which must be reachable in kernel mode. Our technique is based on the fact that in ARM32 CPUs when the MMU is unable to resolve an address it raises an interrupt and it inserts information about the cause of the faulty translation inside the `IFSR` / `DFSR` registers. These are privileged registers accessible only when the CPU is in kernel mode and must be read by the MMU-related interrupt handlers to correctly manage the fault. Since this code snippet needs to always be accessible in memory, we can look for opcodes that read the content of `IFSR` / `DFSR` registers to identify physical pages which must be indexed from a valid kernel radix tree and filter out false positive ones.

4.4.4 ARM (64-bit)

MMU Internals

Introduced in 2011, ARM 64-bit (also known as AArch64) is a new instruction set that shows retro compatibility with ARM32 and, at the same time, drastically changes the internal organization of the CPU by introducing new operating modes, new MMU modes, and by defining different sets of system registers for different CPU modes. The ARM32 `TTBR0`, `TTBR1` and `TTBCR` registers are replaced by `TTBR0_EL1`, `TTBR1_EL1` and `TCR_EL1`: the `EL1` suffix indicates that they are writable when the CPU is in kernel mode and are used in kernel and in the less privileged user mode (suffixes `EL2` and `EL3` are used to indicate registers that are part of the register sets of the higher privileged hypervisor and monitor modes). Note that also the MMU configuration register (`TCR`) is duplicated between CPU modes: the MMU can be configured to have different behaviors when the CPU runs user and kernel code, hypervisor, or monitor one. AArch64 introduces two new MMU modes: the long mode and LPA long mode, the latter using a slightly modified version of table entries derived from the former and supporting a physical address space up to 52 bits. These two modes are based on the long mode of ARM32 and they inherit the impossibility to distinguish directory tables of different levels, thus permitting the existence of loops in the radix tree. Important changes are introduced by AArch64 on the supported physical page sizes and the number of levels of the radix trees. AArch64 support three physical page size (4KiB, 16KiB, and 64KiB, plus huge pages) which can be configured independently for the kernel and the user processes. Furthermore, the sizes of the kernel and user process parts of the virtual address space and the sizes chosen for the physical pages determine the size of the tables, their alignment in memory, and the number of levels in the radix trees.

Analysis

To deal with the AArch64 MMU specificity, we adapted the parsing technique used for ARM32, but replacing the `IFSR` / `DFSR` registers with their corresponding `ESR_EL1`, `FAR_EL1` and `ELR_EL1`. We also modified the algorithm to reconstruct radix trees compatible with the new constraints based on the shape imposed by `TCR_EL1`. The impossibility, also for ARM64, to recover the IDT does not permit to resolve the ambiguity due to the presence of loops in trees, supposedly leaving more false positives compared to ARM32 short descriptor mode. This affects in particular user processes that do not contain any specific instructions or code snippets we can use to filter out false positives.

4.5 Group II: Inverted Page Tables

The POWER architecture is a RISC architecture introduced by IBM in the early '90s. Starting from the original POWER ISA, many architectural variants have been developed during the last 30 years, including variants for gaming consoles. However, only two of those architectures are still currently developed and supported: PowerPC and Power ISA.

4.5.1 PowerPC (32-bits)

MMU Internals

PowerPC is a 32-bit big-endian architecture used mainly in desktop and medium-class servers. It supports only one MMU operating mode based on Block Address Translation + Inverted Hash Table (BAT+SDR1 mode). In BAT+SDR1 mode the MMU uses two concurrent resolution processes [Fre05], one based on block translation and one on the inverted page table: if the block one succeeds the inverted page resolution is interrupted.

The block translation mode relies on a set of CPU registers (the `BAT` set) to define fixed mappings between contiguous blocks of virtual address space to same-size blocks in the physical address space. If the virtual address to be translated belongs to one of the ranges defined by the `BAT` registers the MMU returns the associated physical address. Each `BAT` register describes the characteristics of one of the contiguous blocks: its physical start address, its length, the start address of the associated virtual address space, the permissions flags and CPU modes that are allowed to access the memory.

The inverted page table resolution walk starts, instead, by selecting one of the 16 segment registers (`SR`, in Figure 4.4) according to the upper four bits of the virtual address. Segment registers contain access permission bits in addition to a virtual segment identifier (VSID), which is combined with the remaining part of the virtual address to form the Virtual Page Number (VPN). The VPN is then passed as input to the hash function to get the key for the hash table (whose starting address and size are stored in `SDR1` register). The extracted value, the real page number (RPN), is the physical address of the page associated with the initial virtual address.

Analysis

As in the case of ARM and AArch64, a combination of parsing and data-flow analysis is required to recover the information stored in the CPU registers. Our analysis starts by looking for the

hash tables of the minimum size allowed by the ISA and then we progressively aggregate them to form bigger ones. We also perform a code analysis phase to try to recover the SDR1 value which indicates, without ambiguity, the physical address and the size of the table. The same code analysis also tries to recover the content of the SR and BAT registers. However, some of the BAT registers could be used on a system-wide base, for example by defining fixed blocks of physical memory dedicated to the kernel data and code, or on a per-process base, thus changing at every context switch. In the second case, the code analysis phase cannot succeed because of the custom storage format used by the kernel to save the registers in its private context switch data structures. In the same way, the SR register set is used to define segments inside the segmented address space and, in general, the majority of them change during a context switch. This fact does not permit to recover the segment definitions and cannot permit to assign them, unambiguously, to each different process.

This complexity in the use of the BAT and SR registers affects the ability to recover the virtual address space of the kernel/user processes in two different ways. First, even when we can recover the content of BAT registers, they do not contain any information about which process they refer to. Furthermore, the MMU uses the upper four bits of a virtual address resolved through the inverted page table to choose the segment to use during the translation. Therefore, if we are not able to extract and associate the SR registers to a process, we can only partially reconstruct the virtual to physical mapping. In Section 4.8.4 we will see the consequences of these on the results.

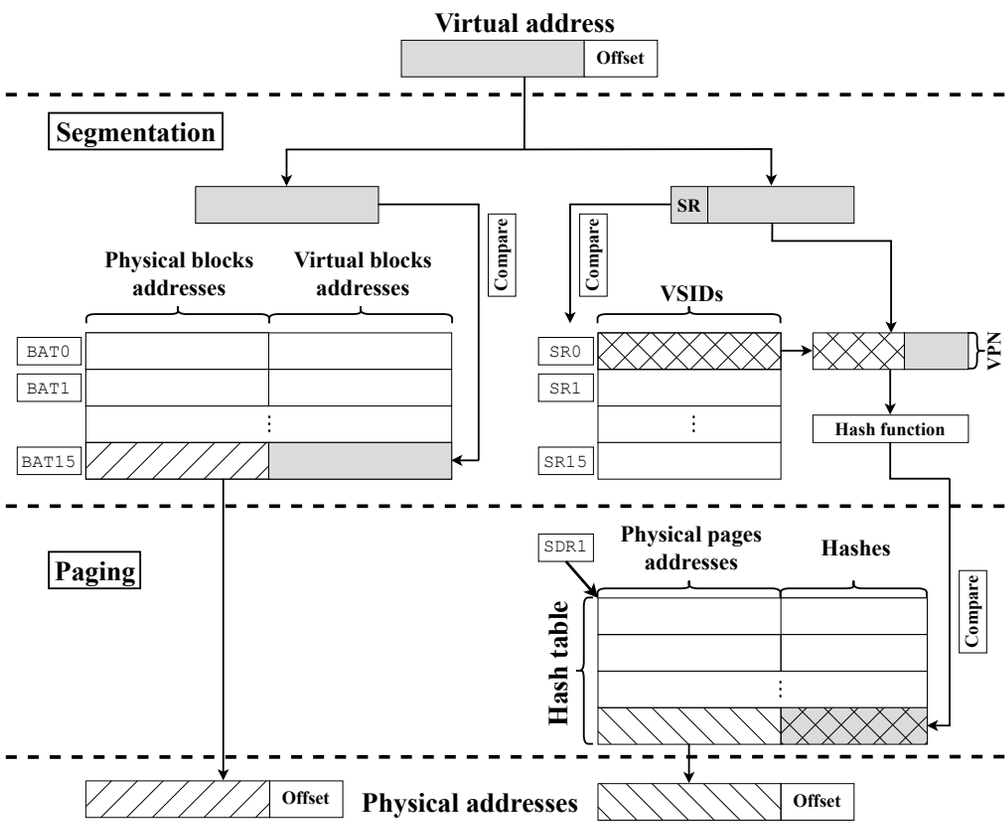


Figure 4.4: Address resolution in PowerPC: block address translation on the left and inverted page table on the right.

4.5.2 Power ISA (64-bit)

MMU Internals

The Power ISA is a server-class 64-bit bi-endian architecture focused on para-virtualization: a hypervisor OS manages all the resources of the system, including MMU registers and in-memory tables, partitioning them between the para-virtualized machines (called Logical Partitions or LPARs). An OS running inside an LPAR needs specific support for this environment in order to interface with the hypervisor. In fact, some system management areas, which in physical systems are directly controlled by the OS, e.g. the MMU initialization and management, in an LPAR environment are under the exclusive control of the hypervisor which manages them for all the LPARs active on the system. CPUs based on the latest Power ISA revision [IBM17], like the IBM POWER9 family, use the `PTCR` register to point to an in-memory hypervisor-reserved table called Partition Table, which contains pointers to relevant memory management structures of the hypervisor and every LPAR running on the system. Each entry of this table points to one of two different types of structures automatically used by the MMU to translate virtual addresses of the LPAR (or the hypervisor): an Inverted Page Table (along with an optional segment table used instead of the segment registers, where not available) or the root of the radix tree of the kernel (with a related process table containing roots of process radix trees).

The Inverted Page Resolution mode works in the same way as for PowerPC (without `BAT` registers) with, in addition, the support for 64-bit virtual address spaces and multiple page sizes. In this mode, all the LPARs share the hash table with the hypervisor calling it to fill the hash table and the segment registers (the optional segment table is filled directly by the OS).

In radix tree mode, the translation of a virtual address to a physical one goes through a double translation: a radix-tree, allocated by the guest OS, translates the virtual address to a pseudo-physical address. Then, a radix tree associated with the LPAR and maintained by the hypervisor, translates the pseudo-physical address to a real physical one. Radix trees have a predefined number of levels, page sizes and table entry format which permits to distinguish between directory and page tables. Structure parameters of this MMU mode as well as the number of tree levels, the size of the paging tables and the size of the physical memory pages are implementation dependent, but all the Power ISA 3.0b compliant processors must support a radix tree composed by 4 paging table levels able to point to 2MB, 64KB or 4KB physical memory pages.

Analysis

On a Power ISA machine we can perform a physical memory dump in two different ways: a dump of the entire machine memory or a dump of a single LPAR. In the first case, we can recover the value contained in the `PTCR` register using code analysis or parsing for the Partition Table using rules derived from the ISA. If an LPAR or the hypervisor uses the radix tree MMU or the hash table with the segment table it is possible to recover the entire virtual to physical address mapping starting from the Partition Table. Instead, if they use the hash table MMU mode with the segment registers set, we can only reconstruct the segmented to physical translation, in the same ways as we described for PowerPC.

Finally, if we work with an LPAR memory dump we can recover the structure of the guest radix trees but if the LPAR uses the hash table mode, we cannot recover any information about

the virtual to physical address translation because the hash table resides in the hypervisor memory.

4.6 Group III: Software-defined Address Translation

In the past, this group included various architectures, such as Alpha and UltraSPARC [JM98], which are not produced anymore: the last remarkable architecture remained is MIPS.

4.6.1 MIPS (32 and 64-bit)

MMU Internals

MIPS is a modular RISC architecture developed by MIPS Technology and largely used in embedded devices. The current ISA release (revision 6) features both 32-bit and 64-bit bi-endian CPU flavors and it supports a range of both mandatory and optional MMU modes. The segmentation unit is always active, also during boot, and the virtual address space is partitioned into multiple segments with different access permissions and translation policies. Segments can be mainly classified in *unmapped* and *mapped*: virtual addresses which belong to unmapped segments are translated directly into physical addresses. On the contrary, mapped segments are paginated and the virtual addresses which belong to them are translated by using the Translation Lookaside Buffers.

The MIPS ISA defines a default set of segments that contains at least one unmapped segment contain the kernel code (`kseg0` and `kseg1` in 32-bit and `xkphys` in 64-bit implementations). The ISA also defines a default mapped segment which contains virtual addresses associated with code and data of user processes (`useg` / `xuseg`), and at least one mapped segment which contains data structures of the kernel (`kseg2` and `kseg3` in 32-bit and `xkseg` in 64-bit implementations).

At boot, the segments use the ISA predefined layout (shown in Figure 4.5) but some MIPS CPU models permit, modifying the content of the MMU `SegCtl` registers, to redefine segments by changing their starting virtual address, size, access permissions and translation mode. They also allow to completely disable the unmapped kernel segment (EVA mode), thus using paging also for the addresses of the kernel code.

All the architectures we discussed so far automatically manage the TLBs content by using the MMU hardware TLB refiller which, thanks to the in-memory structures defined by the OS, can resolve segmented addresses never solved before and refill the TLBs. However, MIPS CPUs by default do not have a hardware TLB refiller and all the complexity of the paging and TLB refilling is therefore delegated to the operating system. This approach offers great flexibility as the OS can choose which data structure to use to manage the memory allocation and segmented addresses translations, e.g. by using a complete software implementation of radix trees or huge arrays of page table entries [4]. When the MMU does not find a valid TLB entry able to translate a segmented address, it raises an exception. In response, the OS needs to look into its software paging data structures and refills the TLB with the missing physical page.

[4]The MIPS architecture uses a dedicated `Context` register to support this last technique as illustrated in [Mip15]. This register is used by the OS to maintain the address of an array of physical pages allocated for the current process, changing its value at each context switch. However, in contrast to other architectures, the resolution of segmented addresses and the refill of the TLBs continue to remain a software task.

Finally, the MIPS ISA defines also an optional MMU mode based on a hardware radix tree TLB refiller, which is much more flexible than the equivalent modes provided by other architectures. In fact, the number of the radix tree levels, the size of huge pages, and the bits of a segmented address that need to be translated are all completely configurable via MMU registers. Furthermore, the format of the page table entries is not completely defined by the ISA, which only lists the essential fields and leaves their absolute positions inside the page table entries to be configured via MMU registers.

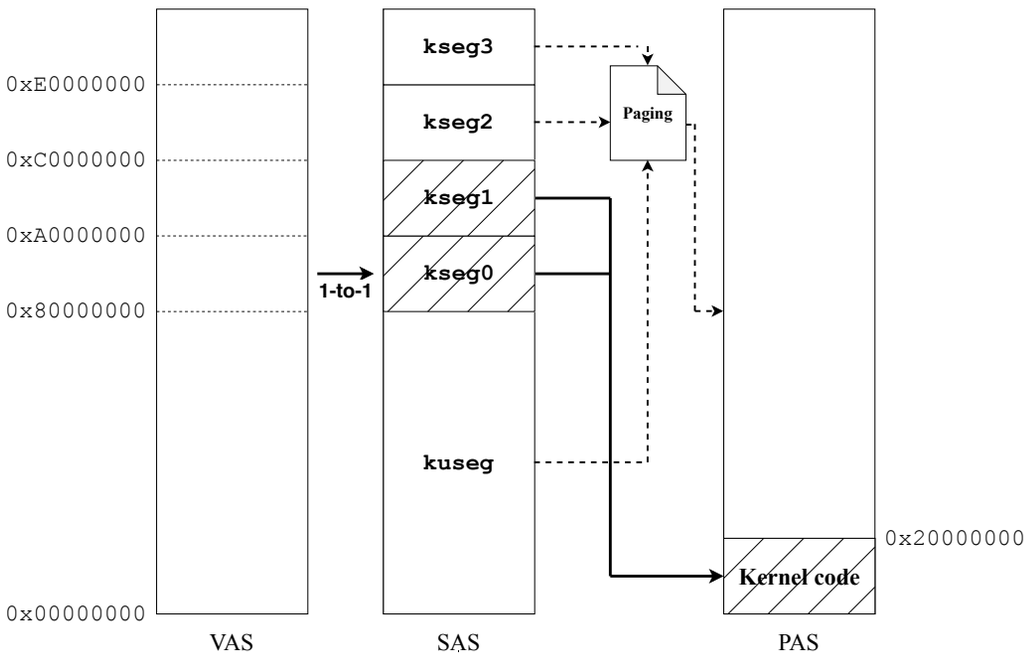


Figure 4.5: Segmentation and paging in MIPS32 using default segments layout. The virtual address space is mapped one-to-one to the segmented one. `kseg0` and `kseg1` segments are translated, using fixed offsets, to the same physical memory chunk. The other segments are paginated: the translation process involves TLBs and hardware-assisted radix-trees translation (if available and configured).

Analysis

The lack of in-memory translation structures (in the case of default MIPS MMU mode), and the great complexity and variability of the page tables (in the case of MMU supporting TLBs hardware refiller), makes it very difficult to implement a fully automated recovery technique. However, we can still use our code analysis phase to recover the configuration of the segments (`SegCtl0, SegCtl1, SegCtl2` registers) and, if the CPU supports it, also the configuration values of the TLBs hardware refiller (`Config, Config5, ContextConfig, PageGrain, PageMask, PWBase, PWCtl, PWField, PWSize, Wired` registers). Unfortunately, even when all register values can be recovered, they can only be used to reconstruct the segment configuration and the code part of the virtual address space of the kernel, which is loaded inside unmapped segments. In EVA mode and for user process virtual address spaces in systems without a hardware TLBs refiller, a manual analysis of the TLB interrupt handler (that can be easily located in a memory dump by

looking for opcodes that load the TLBs) could allow an analyst to reverse engineer the custom algorithm used by the kernel to map virtual to physical pages. Instead, in the presence of a hardware TLBs refiller, by combining the MMU configuration retrieved by the code analysis phase with a custom parser for page tables, it could be possible to manually recover the radix trees used to resolve the virtual address spaces of kernel data and user processes.

4.7 Implementation

In the previous sections, we presented the internal details of how the MMU translates virtual to physical addresses on different architectures. We also discussed how the different structures can be identified and reconstructed from a memory image, and which class of techniques is required to retrieve the different pieces of information. Table 4.1 presents a summary of the various architecture and MMU modes and shows what could be recovered when operating *in a perfect scenario*. In other words, this ignores possible false positives and assumes a perfect code analysis phase that always succeeds in reconstructing the value of the MMU registers. Thus, the results presented in the table are a useful upper bound of what could be achieved in theory. In reality, results can be much worse.

Therefore, to test and validate the techniques discussed in Section 4.3 in real-world scenarios, we implemented them in a proof-of-concept tool written in Python: MMUShell. MMUShell takes as input a memory dump and a YAML file that describes the hardware machine on which the image was collected. This includes information about the CPU architecture, the MMU mode (if known), and the default hardware values of the machine at the reset (e.g. the layout of the physical address space and the initial content of some CPU registers). All these are public information, that can be extracted from the hardware specifications, and are completely independent of the operating system that is running on the device.

MMUShell implements all parsing rules listed in Appendix and uses them to create patterns that can locate candidate MMU data structures. Whenever the system needs to retrieve a value contained in a MMU register, MMUShell identifies all the locations of instructions that operate on the corresponding register and retrieves all previous instructions until it finds an unconditional jump, a long jump, an instruction signaling a function epilogue (e.g `RET` instruction for AArch64 architecture), special instructions (e.g instructions that permit the return from an interrupt handler), or an invalid opcode. This backward analysis is possible because all the architectures for which we need to recover MMU registers have aligned and fixed-size opcodes, thus allowing for a backward linear disassembly.

Results could be improved by extracting the entire function that contains the target instruction (and then also computing a callgraph to reason inter-procedurally). However, the functions executed during the OS initialization, including those that configure the MMU, are often hand-written in assembly and do not have an easily recognizable structure as those produced by a traditional compiler toolchain. In our experiments, this fact prevented automated disassembly and function recognition tools to identify the required functions.

After extracting the code as described above, MMUShell analyzes it to retrieve the values that were written by the OS in the MMU registers. To implement this analysis we evaluated different popular options, which however failed to support all the architectures in our study. Eventually, we decided to extend the Miasm framework [ea23b]. Miasm supports symbolic analysis by first lifting binaries of different architectures to a custom intermediate representa-

Table 4.1: Data structure/register recovery for all the studied architectures *in a perfect scenario*.

Architecture	**MMU mode**	**Mode structure**						**Recoverability**				
		Segmented	Paginated	Paging Mode	In-memory structs	In-register structs	Conf. registers	Kernel MMU structs	User MMU structs	Kernel VA space	User VA space	Access perms
AArch64	Long	○	●	◊	●	○	●	●	●	●	●	◐[1]
	Long LPA	○	●	◊	●	○	●	●	●	●	●	◐[1]
AMD64	4-level	◐[2]	●	◊	●	○	○	●	●	●	●	●
ARM32	Short	○	●	◊	●	○	●	●	●	●	●	◐[1]
	Long	○	●	◊	●	○	●	●	●	●	●	◐[1]
MIPS32	TLB	●	◐[3]	-[4]	◐[4]	○	●	◐[4]	◐[4]	◐[5]	◐[5]	●
	Radix	●	◐[3]	◊	●	○	●	●[6]	●[6]	●[6]	●[6]	●
MIPS64	TLB	●	◐[3]	-[4]	◐[4]	○	●	◐[4]	◐[4]	◐[5]	◐[5]	●
	Radix	●	◐[3]	◊	●	○	●	●[6]	●[6]	●[6]	●[6]	●
Power (System wide)	HTAB	●	●	□	●	●	●	◐[7]	◐[7]	◐[7]	◐[7]	◐[7]
	Radix	○	●	◊	●	●	●	●	●	●	●	●
Power (LPAR)	HTAB	●	●	□	●	○	●	○	○	○	○	○
	Radix	○	●	◊	●	○	●	●	●	●	●	●
PowerPC	BAT+SDR1	●	●	□	●	●	○	◐[8]	◐[8]	◐[9]	◐[9]	◐[10]
RISC-V32	SV32	○	●	◊	●	○	○	●	●	●	●	●
RISC-V64	SV39	○	●	◊	●	○	○	●	●	●	●	●
	SV48	○	●	◊	●	○	○	●	●	●	●	●
x86	IA32	●	●	◊	●	○	○	●	●	●	●	●
	PAE	●	●	◊	●	○	○	●	●	●	●	●

Paging modes: ◊ Radix tree, □ Inverted page table.

[1] Some permission bits depend on SCTLR register value.

[2] Segmentation unit is active but virtually disabled.

[3] Paging is used for kernel data and user address space.

[4] Software dependent.

[5] ISA defines specific address ranges for kernel and user address spaces, internal allocations are software dependent.

[6] Not in an automatic way.

[7] Segment table can be contained inside special TLBs, which in general are not recoverable.

[8] SR and BAT registers can be associated with the running process making them not recoverable using data-flow analysis.

[9] The virtual address space mapping is only partially recoverable.

[10] Permissions for paginated segmented address space are stored in SR registers, which in general are not recoverable.

tion. However, Miasm was developed to operate on user space binaries that do not contain all the special and privileged instructions we encountered in our experiments. So, we had to extend the framework to include both the MMU registers and the main operands used to load and retrieve their values.

MMUShell implements 11 out of 19 MMU modes described in Section 4.3. We did not implement ARM Long (which is almost identical to AArch64 Long) as we could not find OS that supports it and RISC-V SV48 (which is an extension of SV39 and, currently, is not supported by any CPU available on the market). The code analysis phase for MIPS64 is currently not supported by Miasm (however, the functioning of MIPS64 MMU is exactly the same as MIPS32, which is supported by our tool). We have decided to not include Power ISA in our analysis for four reasons: first of all, this architecture is strongly oriented to virtualization and the MMU operating modes are structured with this perspective, opening the problem of the privilege level to whom the memory dump is taken, which influences, in turn, the features' recoverability. Second, QEMU, which we will use to run test case OSs, does not support the emulation of a complete Power machine running an LPAR making it impossible to test the reconstruction techniques in the case of a hypervisor memory dump. Third, Miasm does not support Power ISA, making it impossible to perform the code analysis phase. Last but not least, the only operating system we could find that runs in hypervisor mode is Linux and this lack of test cases would not allow to have any statistical significance on the obtained results.

4.8 Experiments

To evaluate the accuracy and performance of our approach we tested MMUShell on memory dumps running 26 different operating systems with 10 different MMU modes. Each system was configured with 4GB of RAM unless the OS required a different amount (see Tables 4.3-4.7 for more details). We have chosen to limit the amount of RAM to 4GB because many 32bit architectures cannot address more memory and because 4GB also represents a typical memory size that can be found in IoT devices.

To avoid possible inconsistencies [PFB19] and establish a ground truth for our experiments, we run each OS in a virtual machine hosted by a custom version of QEMU [ea23a]. In particular, we modified the emulator to record, for each write operation on MMU registers, the timestamp and the written value. The use of QEMU also allowed us to acquire a copy of the physical memory layout of the emulated machine. For our tests, we installed each OS, booted the machine, and manually used the system by issuing a number of basic commands before acquiring the physical memory.

The operating systems were selected to be as varied as possible in terms of kernel architecture, (including monolithic, microkernel, nanokernel, hybrids, multikernel, and real-time), public availability of the source code, programming languages used to implement the kernel (assembly, C, Rust), year of release (from 1993 to 2020) and purpose (spanning embedded devices, general-purpose, hobbyists project, research, and teaching). The list also deliberately includes some very old (and obsolete) OSs to better prove the generality of our approach. Table 4.2 lists all the OSs along with the architectures in which we were able to run them. Note that Linux is present twice, first as a minimal distribution (Buildroot [bui23]) similar to what one might expect to find in embedded devices, and then with a full-fledged Debian installation. The reason is that we were not able to find a popular desktop distribution that runs on all

Table 4.2: Summary of the operating system analyzed.

OS			Architectures MMU modes									
	Kernel type[1]	Open-source	AArch64 Long	AMD64	ARM32 Short	MIPS32 TLBs	MIPS32 Radix	PowerPC	RISC-V SV32	RISC-V SV48	x86 IA32	x86 PAE
9Front [com23]	H	●	○	○	○	○	○	○	○	○	●	○
Barrelfish [bar23]	U	●	●	●	●	○	○	○	○	○	○	○
Darwin [dar23]	H	●	○	●	○	○	○	○	○	○	○	○
Embox [emb23]	R	●	●	○	●	●	○	○	○	○	●	○
FreeBSD	M	●	●	●	○	○	○	○	○	●	●	○
GenodeOS [gen23]	m	●	○	●	○	○	○	○	○	○	○	○
HaikuOS [hai23]	H	●	○	●	○	○	○	○	○	○	●	○
HelenOS [hel23]	m	●	●	●	●	●	○	●	○	○	○	○
Linux Buildroot [bui23]	M	●	●	●	●	●	●	●	●	●	●	●
Linux Debian	M	●	●	●	●	○	○	●	○	●	●	○
MacOS 9	n	○	○	○	○	○	○	●	○	○	○	○
MacOS X	H	○	○	○	○	○	○	●	○	○	○	○
Minix3 [min23]	m	●	○	○	○	○	○	○	○	○	●	○
MorphOS [mor23]	m	○	○	○	○	○	○	●	○	○	○	○
NetBSD	M	●	●	●	●	○	○	●	○	○	●	○
Illumos [Edi23]	M	●	○	●	○	○	○	○	○	○	○	○
QNX [qnx23]	R	○	○	○	○	○	○	○	○	○	●	○
rCore [rco23]	M	●	●	●	○	○	○	○	●	●	○	○
ReactOS [rea23]	m	●	○	○	○	○	○	○	○	○	●	○
RedoxOS [red23]	m	●	○	●	○	○	○	○	○	○	○	○
vxWorks [vxw23]	R	○	○	●	○	○	○	○	○	○	○	○
Windows 10	H	○	●	●	○	○	○	○	○	○	○	●
Windows 95	M	○	○	○	○	○	○	○	○	○	●	○
Windows NT	H	○	○	○	○	○	○	○	○	○	●	○
Windows XP	H	○	○	●	○	○	○	○	○	○	●	●
XV6 [xv623]	M	●	○	○	○	○	○	○	○	●	○	○

[1] H: hybridkernel, m: microkernel, M: monolithic kernel, U: multikernel, n: nanokernel, R: real-time kernel

our architectures and MMU configurations. While Debian is supported only by a subset of the Linux Buildroot configurations, it has one order of magnitude more processes (and therefore more MMU structures) to recover.

The MMU registers values collected by the emulator are used as ground truth. However, it is necessary to filter out those obsolete entries that reference data structures not present anymore in RAM at the time we acquired the memory dump (e.g. radix trees of defunct processes freed by the kernel). While it might still be possible to recover these data structures

Table 4.3: RISC-V (32 and 64 bit)

OS	MMU mode	VAS	FN	FP
FreeBSD	SV39	9	-	-
Buildroot[1]	SV32	7	-	-
Buildroot	SV39	7	-	-
Debian	SV39	18	-	-
rCore	SV32	2	-	-
rCore	SV39	2	-	-
XV6	SV39	3	-	-

[1] Does not support 4GiB of RAM.

from memory, their bytes might have been already partially overwritten by other processes, thus making it difficult to locate and validate their correctness. For this reason, we performed a self-consistency check on radix trees and remove those that contained at least one table that is no more valid. We also verified the last time the kernel wrote a value into each MMU register to identify only the last configuration used.

In our proof of concept implementation, we did not pay particular attention to the performance, also because our analysis is a one-time operation that needs to be performed only once. According to our experiments, the slowest operation (the search of IDT tables) required approximately 1 hour for a 4GB dump on a 4 core x86 CPU.

In the rest of the section, we discuss in detail the results for the three groups of architectures.

4.8.1 Group I: RISC-V 32 and 64-bit

Table 4.3 shows the results for RISC-V 32 and 64 bit. In particular, for the various operating systems and MMU modes, the table reports the total number of virtual address spaces (VAS) retrieved from the modified QEMU (i.e., the ground truth). The last two columns show the number of false negatives (i.e., the address spaces that could not be correctly reconstructed by our tool) and false positives (i.e., spurious data structures that were erroneously reported as address spaces).

In this case, MMUShell was able to successfully retrieve all the correct radix trees associated with kernel and user space processes without any false positives. Therefore, our system could fully reconstruct the virtual-to-physical memory mapping for the kernel and each running process.

4.8.2 Group I: Intel x86 and AMD64

The results for Intel x86 and AMD64 are reported in Table 4.4. In this case, the tables include two additional columns to report whether the correct Interrupt Descriptor Table (IDT) has been located in the dump and the number of additional candidates IDTs. While the extraction of the IDT is not the goal of our analysis, it plays a very important role in the results because, as we explained in Section 4.4.2 MMUShell uses this information to validate the candidate page tables. In fact, when MMUShell was not able to retrieve a valid IDT, the number of false positives rose considerably. The failure in IDT recovery is due to the high variability of possible configurations of the interrupt descriptor fields set by the various OSs, and to the impossibility

Table 4.4: INTEL x86 and AMD64

Intel x86 IA32 / PAE MMU modes						
OS	**MMU mode**	**IDT**		**VAS**	**FN**	**FP**
		Found	FP			
9Front	IA32	●	3	61	-	8
Embox	IA32	○	-	1	-	54
FreeBSD	IA32	●	1	21	-	-
HaikuOS	IA32	●	1	18	-	12
Buildroot	IA32	●	1	9	-	1
Buildroot	PAE	●	1	10	-	1
Debian	IA32	●	1	68	-	2
Minix3	IA32	●	-	209	-	2
NetBSD	IA32	●	-	18	-	-
QNX	IA32	●	2	40	-	1
ReactOS	IA32	●	1	17	-	-
Windows 10	PAE	●	5	137	-	34
Windows 95[1]	IA32	●	-	1	-	-
Windows NT	IA32	●	-	14	-	-
Windows XP	IA32	●	-	19	-	-
Windows XP	PAE	●	-	19	-	1

[1] Does not support more than 512MiB of RAM.

AMD64 4-level MMU mode					
OS	**IDT**		**VAS**	**FN**	**FP**
	Found	FP			
Barrelfish	●	-	31	-	-
Darwin	●	-	16	-	11
FreeBSD	●	2	48	-	3
GenodeOS	○	-	61	-	68
HaikuOS	●	-	17	-	1
HelenOS	○	-	63	-	63
Buildroot	●	-	22	-	2
Debian	●	-	71	-	2
NetBSD	●	-	18	-	-
OmniOS	●	6	4	-	-
rCore	○	1	3	3	-
vxWorks	○	-	2	-	-
RedoxOS	○	-	62	-	33
Windows 10	●	1	127	-	4
Windows XP	●	2	21	-	-

to predict them by using only OS-agnostic information.

MMUShell was able to find all the virtual address space in all the tested configurations with the exception of rCore running on AMD64, for which it wrongly identifies the IDT. Therefore, even if the correct radix trees were found by the tree reconstruction part of the algorithm, they were later discarded as possible FP because they could not resolve the virtual addresses of the false interrupt handlers.

4.8.3 Group I: ARM32 and AArch64

For ARM32 and AArch64 our tool needed to retrieve the part of the `TTBCR/TCR_EL1` register (`N` and `EAE` for ARM32, `T0SZ, T1SZ, TG0, TG1` for AArch64) that controls the shape of the processes radix trees in ARM32 and kernel and user ones in AArch64. In Table 4.5 and Table 4.6 we report the statistics of the retrieved register fields values for each OS and then report the radix trees with a shape compatible with the true positive values of `TTBCR/TCR_EL1` fields. For ARM32 the data flow analysis permitted, for all the OSs, to retrieve both the values of fields of `TTBCR` register as well as all the kernel radix trees and, when `TTBR1` is used, also all the user process ones.

For AArch64 instead, in two cases (Embox and Windows 10) it was not possible to recover the four `TCR_EL1` fields necessary to reconstruct the radix tree and virtual address space shape. In another case (Barrelfish) we failed to retrieve only one of these fields. For all the tested configurations MMUShell recovered all the radix trees for kernel and user space processes.

Table 4.5: ARM

ARM32 Short MMU mode								
OS	**TTBCR fields**		**Kernel VASs**			**User processes VASs**		
	TP	FP	VAS	FN	FP	VAS	FN	FP
Barrelfish	2/2	1	1	-	6	26	-	6
Embox	2/2	-	1	-	2	-[2]		
HelenOS	2/2	-	33	-	6	-[2]		
Buildroot[1]	2/2	-	14	-	1	-[2]		
Debian[1]	2/2	-	77	-	1	-[2]		
NetBSD	2/2	1	1	-	-	15	-	-

[1] `vexpress-a9` machine supports a maximum of 1GiB of RAM.
[2] Uses only `TTBR0` due to `TTBCR.N=0` [Hol18]

Table 4.6: AArch64

AArch64 Long MMU mode								
OS	**TCR_EL1 fields**		**Kernel VASs**			**User processes VASs**		
	TP	FP	VAS	FN	FP	VAS	FN	FP
Barrelfish	3/4	1	1	-	1	10	-	-
Embox	0/4	-	1	-	1	-	-	-
FreeBSD	4/4	-	1	-	3	14	-	-
HelenOS[1]	4/4	-	2	-	2	49	-	-
Buildroot	4/4	2	1	-	1	5	-	-
Debian	4/4	2	1	-	1	15	-	-
NetBSD	4/4	1	1	-	1	16	-	-
rCore[2]	4/4	-	2	-	-	1	-	-
Windows 10	0/4	-	107	-	1	97	-	2

[1] Does not support more than 1GiB of RAM.
[2] `raspi3` machine supports a maximum of 1GiB of RAM.

Table 4.7: PowerPC and MIPS32

PowerPC BAT+SDR1 MMU mode						
OS	**BAT registers**		**Hash Table**			
	Subregs.	BAT	Location	Size	FP	SDR1
HelenOS[1]	8/16	0/8	●	●	-	○
Buildroot[2]	16/16	8/8	●	●	-	●
Debian[2]	16/16	8/8	●	●	-	●
MacOS 9[3]	12/16	4/8	●	●	-	○
MacOS X[2]	-[4]	-	●	●	-	●
MorphOS[2]	6/16	3/8	●	●	-	○
NetBSD	7/16	2/8	●	●	6	○

[1] `g3beige` machine supports a maximum of 1GiB of RAM.
[2] `mac99` machine supports a maximum of 2GiB of RAM.
[3] Does not support more than 1GiB of RAM.
[4] See text for more details.

MIPS32 TLBs / Radix Tree MMU modes				
OS	**MMU mode**	**MMU registers**		
		TP	FN	FP
Embox	TLBs	13	-	-
HelenOS[1]	TLBs	13	-	2
Buildroot[2]	Radix	12	1	6
Debian[2]	TLBs	13	-	2

[1] `malta` machine supports a maximum of 2GiB of RAM.
[2] Does not support more than 256MiB of RAM on `malta` machine.

4.8.4 Group II: PowerPC 32-bit

In the PowerPC architecture, the address resolution process involves the use of the `BAT` registers and an inverted page table pointed by the `SDR1` register. Each `BAT` register is physically implemented as a couple of 32-bit sub-registers (`BATU` and `BATL`) which are loaded separately by the CPU but used as an atomic unit by the MMU. The `BATU` subregister contains the address of the translated virtual address block and its size, while the companion `BATL` contains the physical address of the correspondent block plus its access permissions. Thus, even if only one of the two subregisters is automatically retrieved from a memory dump, it still provides important information to the human analyst about the layout of the virtual or the physical address space. For this reason, Table 4.7 reports the number of physical subregisters recovered by MMUShell and the number of complete couples, corresponding to the entire `BAT` register.

Our code analysis phase was able to completely recover the content of the `BAT` set for Linux, partially for Morphos MacOS 9 and NetBSD, and only isolated subregisters for HelenOS. This is due to the lack of interprocedural analysis and the incomplete support of PowerPC MMU instructions in the Miasm framework.

It is important to note that MacOS X defines different address blocks for every process, thus changing the content of the `BAT` registers at every context switch. As a result, the values cannot be recovered without an in-depth knowledge of the data structures used by the OS kernel. On the contrary, MMUShell can retrieve, for every tested OS, the physical address and size of the hash table and, in some cases, also the content of the `SDR1` register. Thus, MMUShell could fully reconstruct, for each OS, the mapping between segmented and physical address spaces, but was not able to completely reconstruct the mapping between the virtual and the segmented spaces. Furthermore, in the case of Linux and partially of MorphOS, MacOS 9 and NetBSD, by retrieving also `BAT` registers, MMUShell could also list the blocks of virtual address space directly mapped to the physical ones.

4.8.5 Group III: MIPS 32-bit

As discussed in Section 4.6.1 for MIPS32 we need to retrieve the partial content of 13 different MMU registers to identify uniquely the MMU configuration of the machine. However, these registers contain also bitfields dedicated to the configuration of other subsystems (e.g., the `MSAEn` field in the `Config5` register controls the SIMD subsystem [Mip15]). Since we are only interested in reconstructing the virtual memory, Table 4.7 reports as a correct result the cases in which MMUShell was able to recover correctly all fields relative to the virtual memory subsystem, ignoring the others. While MMUShell was very accurate in retrieving those values, as explained in Section 4.6.1, these registers provide only half of the picture and a human analyst still needs to manually reverse engineer the algorithm used by the kernel to map segments to physical pages.

4.9 Application to Real Hardware

In order to demonstrate the usefulness and validate the functionality of MMUShell in a real-world scenario and demonstrate how our solution can help in a real investigation, we have conducted a forensic analysis on a real hardware device running an uncommon OS. For this experiment, we chose a Raspberry PI 3B+ [ras23], an AArch64 board that also supports ARM32 and for which we were able to acquire the raw memory content by using the integrated JTAG controller. On the board, we have installed RISC OS [ris23], a partially-open source operating system used in digital TV decoders and factory automation systems [ris23]. This is an excellent example of a popular but not well-known OS that runs on embedded devices but which is not supported by QEMU (and this is why we could not include it as part of our controlled experiments).

By using only the board documentation, we created the necessary YAML file that describes the physical memory layout and the initial default value of the CPU registers. Moreover, through the JTAG interface, we are also able to dump the value of the MMU registers at runtime, and later use their content as ground truth to validate the results automatically extracted from the memory by MMUShell.

MMUShell was able to correctly recover all MMU registers values (as confirmed by our ground truth values collected during the dump phase) and reconstruct the kernel/user radix tree without any false positive (RISC OS, as other real-time OSs analyzed in Section 4.8, uses only one radix tree for all the processes running in the system).

As a result, the analyst now can easily reconstruct the mapping between the physical address space and the virtual address spaces of the kernel and the user processes. This permits to correctly resolve the pointers contained in the dump and allows to start the analysis of the data structures contained in it.

It is important to stress again that MMUShell results derive only from the memory dump and the public information on the hardware architecture. No OS-specific rules or information is necessary for our analysis.

4.10 Towards OS-Agnostic Memory Forensics

Now we want to show a possible use of our technique to perform a preliminary memory analysis of a dump acquired from an unknown operating system. It is important to stress that there are no tools or techniques available today that can operate in these settings. While our tool is only the first step towards an OS-agnostic analysis, by using our virtual memory reconstruction technique we can already automatically extract all running processes and derive information about them without any knowledge of the OS internals, executable file format, or mechanism used to interact with the kernel.

In fact, MMUShell can export each virtual address space as a self-contained ELF core dump file, which maps each physical memory page to its corresponding virtual one inside a `LOAD` segment – along with the page access permissions. This allows an analyst to obtain a faithful representation of the process virtual address space. In addition, each ELF can also be easily opened and analyzed by using popular static analysis tools – such as Ghidra. Each ELF generated by MMUShell corresponds to a process running in the system (either kernel, privileged services, user space programs, or dead processes). This can already be used by a security analyst to retrieve the process running in any IoT device, independently from the architecture or the running OS. The list of running processes in a compromised device could be easily compared with that acquired on a clean system to detect anomalous processes that might need to be manually reverse-engineered.

By using the extracted information, in the rest of the section, we discuss which other properties could be identified for either a single or an ensemble of processes. Unlike the other part of our work, to perform this task we rely on a set of heuristics based on the usage of the MMU by a generic OS, which are reasonable in multiprocess systems with an MMU.

- **Kernel and privileged services identification** – The access permissions of the pages in the process virtual address space allows to discriminate whether a process is a privileged one (which runs at kernel-level and does not permit, in general, any write access to its space to user mode processes or does not map any user space accessible pages) or a user space one (which can have kernel pages mapped in its address space but without write access).

- **Executable file format identification** – It is possible to use tools, like `binwalk` [Hef23], which permit to identify a known executable file format headers (e.g., ELF, PE, COFF, etc.) in the virtual address space of a process, thus recovering additional information about the structure of the executable file loaded supported by the unknown OS.

- **Core shared libraries and data pages** – Modern OSs that support multiple processes and virtual memory usually rely on a special library to support the executable loading process before starting its execution (e.g. `ld-linux.so` in Linux) or core libraries and data pages (e.g. `libc.so`) to easily interface the user space process with the kernel through syscalls. To reduce memory consumption, physical pages that contain core shared libraries are present in a unique copy that is mapped, without write permissions, into the majority of the user space virtual address space. We can find these shared resources by looking for regions in each user process that point to the same physical pages and which are present in at least 90% of the user processes.

- **Minor shared libraries, shared code pages, and interprocess shared memory regions** – By using the same technique without the 90% threshold we identify less common shared libraries, pages of code shared among processes (due to syscall similar to the UNIX `fork`) and, including also pages with permission `RW-`: shared regions used by multiple processes to efficiently share data among them.
- **Special regions** – By using our data, we can identify a set of virtual address ranges that are always mapped in all processes. These (e.g., the `vsyscall` pages in Linux). could play a particular role in the target OS architecture.
- **Stack identification** – The stack pages of a process must have at least `RW-` permissions (in general, for security reasons, executable permission is missing but in some architecture, such as x86-32, this is not possible) and they must be contiguous. Furthermore, if we assume that the stack frame pointer register is used to track old previous stack frame start, stack pages contain adjacent couples of values composed by an instruction address and a stack address. This pattern is directly related to the functioning of stack-based architectures: a `CALL` instruction to a routine pushes the address of the immediately following instruction on the stack (the return address), followed by the address of the previous stack frame contained in the stack frame pointer register. Stack pages can be identified by looking for pages with the correct permissions set and which contains the biggest number of valid couples of return addresses and stack frame pointer addresses. It is possible to identify return addresses because they are virtual addresses that belong to code pages (`R-X`) and tools like `radare2` [ea23c], which can load our virtual address space due to its representation as ELF core file, permit, also in case of non-fixed length opcode architecture like x86, to perform backward decompilation and check if the previous instruction pointed by the return address is a real `CALL` instruction removing a great number of false positives. At the same time, stack frame pointer addresses are supposed to belong to the same page on which they reside (supposing the stack frame is not bigger than a page size or astride them).
- **Approximated entry point** – Directly related to stack identification, we can derive an approximated position of the entry point of the executable: the bottom of the stack must contain the address of an instruction belonging to the first function executed by the program or by the loader library, permitting to identify approximately which part of the code could contain the real entry point.
- **Processes relations** – A manual analysis of which processes share writable memory areas and code areas permit to identify relations among processes. A multiplicity of stack candidates (if we suppose that they are not false positives) may indicate the presence of different threads associated with the process: they will share the same code pages but each will have its own private stack. Finally, it is possible to calculate a TLSH [tls23] fuzzy hash on the content of the single physical pages of two or more processes to check similarities in code and cluster them also in case of code relocations as shown in [PDB18].

4.10.1 Experiment

As an example, we have chosen to analyze the physical memory dump of an x86-32 installation of the QNX operating system. The x86 architecture has been chosen because, as discussed in

previous sections, radix-tree based architectures (such as INTEL, ARM, RISC-V and partially Power) permit a complete reconstruction of the virtual address spaces. The chosen architecture, however, partially complicates our analysis because does not provide any bit to set a page as executable, so all the accessible pages in user mode are at least readable and executable.

We choose the QNX OS because it is a closed-source operating system that does not have a lot of public information about its internals and it was poorly studied by the memory forensics community.

As ground truth, we have extracted, through `/proc/PID/as` interface of QNX, information about the virtual memory space layout of all processes running in the system and their associated threads. MMUShell, as shown in Table 4.4 correctly identified all the valid virtual address spaces present in the memory dump (40), including three virtual address spaces belonging to processes that already terminated (at dump time only 37 processes were running on the system).

By using the heuristics described above, our tool was able to automatically distinguish all processes in user processes (36) from the kernel itself (1) and determine that QNX uses the ELF format. It also identified two core libraries mapped by most processes (which correspond to `ldqnx.so.2` and `libc.so.3`), as well as many other libraries used by a subset of the user space programs and writable memory regions shared among different processes. For 34 processes, our system correctly identified the stack and the instructions belonging to the first function executed by the programs. Furthermore, the presence of multiple stacks on some processes confirmed their multithread nature.

In summary, the analysis and experiment presented in this section show a possible practical use of our tool to perform a preliminary memory analysis for any operating system currently unsupported by existing forensic tools.

4.11 Limitations and Future Extensions

We now discuss some limitations of our approach, either due to engineering problems or to the OS-agnostic nature of our techinque. We believe this list can also provide some ideas for possible extensions and future research in the field:

- Even after our extensions, Miasm still lacks support for several opcodes, in particular for PowerPC and MIPS. This further emphasizes the need of our community for a binary analysis framework able to operate in a symbolic way on privileged code for non-x86 architectures.

- On some architectures, like Intel x86/AMD64 and ARM32/AArch64, when a page table entry has the present bit unset, the MMU simply ignores the content of the remaining bits and raises an interrupt if the CPU tries to access virtual memory addresses potentially resolved by it. However, as extensively shown in numerous works, OSs like Windows, Linux and OSX fill ignored bits of invalid page table entries with values that efficiently permit them to manage extensions to the traditional model of virtual address space. Examples of this behavior are the classical swap to secondary storage, or the compressed swap stores and transition pages implemented in Windows [SA19, Kor07], the `PROT_NONE` pages in Linux [Lev15], OSX memory queues [CMMRI20], or the possibility for Linux and OSX to compress at runtime part of the RAM content [RIC14] to increase the memory

available to the system. All these techniques save a reference to the original page and additional metadata (e.g. an offset within a swap file) into the invalid page table entry relative to the page itself. When the CPU tries to access a virtual address that belongs to those pages, the MMU raises an exception and the OS checks the invalid bits in the entry. If the OS determines, for example, that the page is part of a compressed memory region, it decompresses the page, restores the association between the page table entry and the page, set the valid bit, and restart the execution of the process that has generated the previous illegal access.

Our technique is completely blind to these types of behavior, which is strongly OS-dependent. However, these extensions do not break the inviolable MMU constraints since they are based on the use of ignored bits in invalid page table entries. To access to this type of pages a partially OS-independent approach could consist in the identification of the page fault handler in the interrupt table and its emulation by using the faulting page table entry as a parameter (as specified by the CPU ISA). This automated approach may allow to gain some information about pages marked as invalid but instead treated in special ways by the OS. As a last resort, a human analyst could perform a manual analysis of the page fault handler to understand how it works and how to manage invalid page entries. Furthermore, in the last few years, the memory forensics community has deeply studied the memory management subsystems of Windows, Linux and OSX and has implemented inside tools, such as Volatility and Rekall [VS19], modules able to treat correctly invalid page table entries, thus drastically increasing the analysis capabilities of these systems.

- We found very little information that can be used to build validation rules to reduce the number of false positives. However, in our experience, human experts can often tell the difference between real virtual address spaces and false-positive ones, suggesting that maybe a machine learning classifier can be trained to reduce the false positives. This is an interesting research direction that we leave as future work.

- Even though we performed all our experiments in a 'benign' scenario, as we have specified in various parts of the paper the MMU requires that the OS strictly respects a number of inviolable constraints. A rootkit (or other forms of a compromised system) is no exception to this rule and cannot bypass our MMU constraints.

  However, even though we did not encounter this case in any of the 26 OSs we tested, it is possible that the code responsible to configure the MMU is intentionally deleted from memory after the setup is complete. In this (possibly adversarial) setting, bridging the semantic gap may become very difficult, if even possible at all.

- In our work, we assumed that the OS under analysis is running on a bare-metal machine or inside a VM for which we can get a memory dump containing only the VM memory. However, it is also possible to consider the possibility in which the OS is running as a hypervisor for other VMs. In this case, a memory dump of the full system will contain also the VMs memory. All the treated architectures in this work (but PowerPC and MIPS32) support virtualization in various forms by using, in general, an extension of the already presented MMU modes. In particular, radix-tree architectures extend the existing MMU modes introducing radix-trees composed of a different type of page tables associated

with each VMs. Each of these new radix-trees maps the VM physical page addresses, valid inside the associated VM, to physical pages exiting on the bare-metal machine. An interesting development of our technique could support also the reconstruction of the MMU hypervisor in-memory data structure associated with each single VM which only requires to define new structure signatures to match this new type of data. The special case of Intel x86, memory forensics of hypervisor has also been covered by Graziano et al [GLB13].

4.12 Discussion

In this chapter, we discussed the broad range of strategies adopted by the MMU of different CPU architectures to translate virtual to physical addresses. We have presented the first complete study of the problem, and our solution to bridge the semantic gap of virtual-to-physical address translation in a *zero-knowledge scenario*. We have also highlighted that, due to the peculiarities of some architectures (PowerPC, Power and MIPS), a full automatic complete reconstruction of the virtual address spaces without a deep knowledge of the OS internals is, in the general case, impossible. This fact is not a limitation of our technique but an intrinsic limitation imposed by those ISAs.

We have shown that our solution, based on a combination of OS-agnostic MMU derived constraints (to identify candidate data structures) and static code analysis (to retrieve the setup of system registers) can extract the maximum amount of information, permitted by the CPU ISA, about the virtual address spaces of the running processes. The technique does not use *any* knowledge on the running OS or tailor-made heuristics.

MMUShell, the proof-of-concept tool implementing our technique, was able to recover all the virtual address space in the vast majority of the experiments and also on a dump extracted from a physical hardware device, proving the feasibility of our approach in real-world scenarios. A simple OS-agnostic memory forensics analysis was conducted on a close-sources OS to show the potential of our tool in real-world scenarios.

The amount of time required to add the support for a new architecture would largely depend on the type of virtual-to-physical address translation strategy implemented by its MMU, the analyst knowledge of its internal details, and the quality of the available documentation. For instance, it took us only a few days to support RISC-V (to read the documentation, compile a list of constraints, create a valid strategy to reconstruct radix-trees, and perform some tests) while MIPS and ARM64 required roughly a few weeks each, due to the variability of the available options, complexity and peculiarity of their MMU modes.

Chapter 5

From Virtual Address Spaces to Kernel Data Structures

This chapter is extracted from "An OS-agnostic Approach to Memory Forensics", presented at 30th Network and Distributed System Security Symposium (NDSS) 2023 [ODB23]

5.1 Introduction

The analysis of volatile memory is gaining more and more importance as part of digital forensics and incident response investigations. This is because the system memory contains artifacts, related both to the current and the past state of the system, that cannot be extracted from any other component. However, the analysis of volatile memory presents unique challenges, as operating systems (OSs) have diverse internal organization ways they store and represent data, with only a few constraints imposed by the underlying hardware and memory management unit (MMU). Unfortunately, this great flexibility becomes a great challenge for forensic analysis tools.

For example, the kernel needs to keep track of all processes running inside the system, including their names, memory layout, loaded libraries, and open file descriptors—just to name a few among the many pieces of information required to track processes during their execution. OSs organize this set of information by using data structures, such as linked lists, trees, and arrays. To extract this data from memory dumps and reconstruct the aforementioned structures, analysts must know the (often undocumented) kernel internals to locate and interpret the raw bytes in the dump. This, which is often called the *semantic gap*, constitutes the main problem of memory forensics.

To solve this problem, forensics tools like Volatility [Wal17] and Rekall [Coh14] maintain collections of precise descriptions of the kernel data structures used by popular OSs (e.g., Windows, Linux, and macOS). Thanks to these OS-specific models, called *profiles*, these tools can extract relevant forensics information from memory dumps. However, profiles need to be constantly updated because the internal data structures used by OSs often change with version updates. Furthermore, for highly configurable OSs like Linux, it is often necessary to create specific profiles for the individual machines that analysts want to investigate. While recent works [PB21, QQY22] have partially overcome this problem by reconstructing Linux profiles

directly from the memory dumps, these solutions are tailored to the data structure of the Linux kernel–which are supposed to be known in advance.

Memory forensics is a very active research area and has largely improved over the last decade. However, it still relies on manual rules that exist only for a handful of OSs. While this was sufficient in the past, today investigations often involve a broad range of devices, ranging from network routers to smart home appliances and smartwatches, all of which operate on a variety of OSs. For instance, recently *Cekerevac et al.* [CDP20] studied 56 different OSs adopted in the IoT space alone (a list that does not even include OSs used in network appliances, such as CISCO IOS). Sadly, *most* of these systems are currently unsupported by memory analysis tools. Even worse, the current approach is based on a long and tedious human effort to reverse engineer the required OS internal details; this makes extending memory forensics support to a large number of OSs completely impractical.

5.1.1 Introducing OS-agnostic Memory Forensics

Currently, available forensics tools can extract information from a memory dump by relying on two techniques: (i) the OS is manually analyzed by the tool writers [Coh14, Wal17], who manually compiled a set of rules to extract the forensics-relevant information for that OS (ii) the OS is instrumented (e.g., by analyzing its source code [CMR10, FPW+16, LRW+12, LRZ+11] running it on instrumented VMs [DGSTG09, LZX10], or by taking snapshots in different states [SYLS18, UGCL14]). The first rule-based approach, which is the standard among memory forensic tools, requires months of work to add support to a new operating system. The second solution, proposed by researchers but not commonly used in practice, requires instead complete and privileges access to the target systems. None allows an analyst to recover any information from a memory dump extracted from a device running an OS that is either unknown (no source code, no installation disks or the possibility to run it on a VM) or unsupported by ordinary forensics tools. This situation can occur when, for example, memory is acquired from IoT or industrial devices, network appliances, or VMs that are running uncommon OSs. This is why memory forensics today does not exist for such cases.

To solve this problem, we propose a paradigm shift in the way we perform memory forensics. While profiles, custom rules, and dynamic introspection remain valuable solutions that are likely to provide better results when they can be applied, we believe a new approach is needed to quickly extend memory analysis to a broader class of target systems. If today the field is driven by a *complete knowledge* of the internal data structures, we propose instead the first step towards what we call *OS-agnostic memory forensics*, which would allow analysts to perform memory forensics without *any* information about the underlying OS.

Our approach relies only on OS-agnostic properties of the data structures used by kernels to organize data. In fact, like all software, OS kernels store information in a collection of data structures that can be detected and reconstructed due to their topological properties. As shown in Chapter 4, for most CPU architecture it is possible to reconstruct the kernel address space in an OS-agnostic way by using only information derived from the hardware configuration of the machine. It is by using this information that we reconstruct in-memory kernel data structures like linked lists and trees.

In this chapter, we first propose new algorithms to identify forensics-relevant data structures in a fully-automated way. As forensics-relevant we consider all those data structures which contain, or point to, the information needed during a forensics analysis, thus excluding data

structures that are used by the OS only for its basic functioning (e.g., data structures for hardware management, communication among its internal parts, synchronization, etc.). Our technique takes as input a single memory dump and a function to extract pointers from raw data. It then, without human intervention, extract forensics-relevant data structures based on their topological properties. We investigate two complementary approaches to perform this task. In the first, we start by observing that the vast majority of data structures are irrelevant from the point of view of a forensics analysis. Forensic-relevant data structures can be identified because they reference or embed specific *seeds*, which are pieces of information that an analyst knows *a priori* or can be easily carved from the memory. Examples of seeds are the name of a process, an IP address, or a file name. Seeds are important because the analyst can use them as "anchors" to filter data structures and then automatically extract information of the same type. For instance, knowing just one process name can be sufficient to (i) identify a data structure (such as a doubly-linked list) that points to it, and (ii) explore all the elements of that structure and print the strings that are referenced by pointers located at the same offset in the data structure. Even with no previous knowledge of the OS, this automated procedure allows a forensic analyst to list all process names–once one of them is known.

The second approach we explore is a seed-less analysis. Once again with no prior knowledge of the OS, we show it is possible to retrieve structured forensics information from a memory dump to be used as a bootstrap for advanced analysis. For instance, by filtering and ranking the extracted data structures according to the statistical properties of their fields, an analyst can use our system to look for data structures referring to forensics-relevant strings. We will show that this approach can retrieve most of the data structures retrieved with the first approach, with the advantage of being applicable when analysts do not known valid seeds.

We implemented our techniques in a proof-of-concept tool, *Fossil*, and we performed experiments to retrieve data structures from 14 different operating systems. Our results show that our OS-agnostic approach to memory forensics can provide invaluable information to analysts facing memory dumps taken from less known, or unknown, OSs. We will release Fossil as an open-source project [ODB23], together with the part of the dataset not covered by particular OS license restrictions.

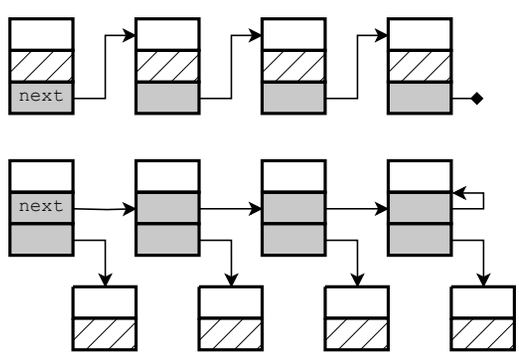


Figure 5.1: Two examples of different implementations of a linked list.

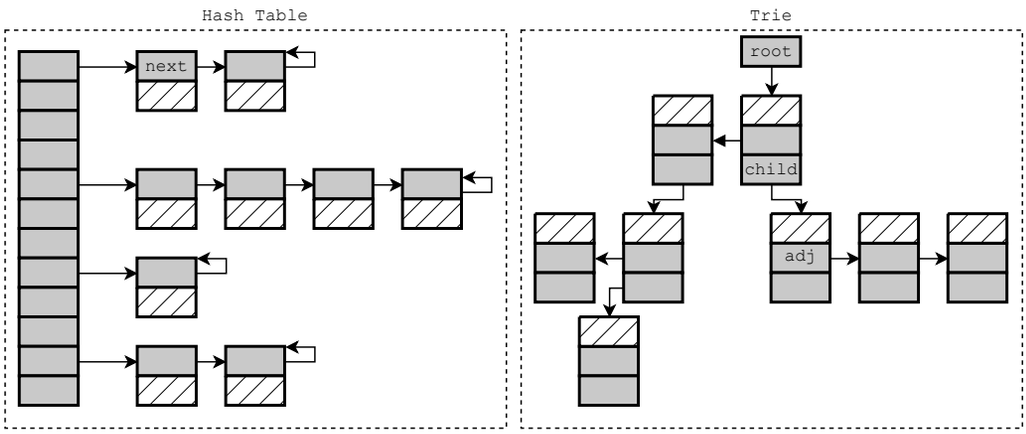


Figure 5.2: Two examples of complex data structures: a hash table and a trie implemented using arrays and linked lists. Grey cells are pointers while barred ones contain data.

5.2 Data Structures in Memory Forensics

Kernels organize information in a variety of data structures (DSs) which can, in turn, be implemented in many different ways. data structures like linked lists, doubly-linked lists and n-ary trees are all based on logical relations among components linked through pointers. On the other hand, structures that require fast access like arrays, stacks and queues are based instead on data locality. As shown in Figure 5.1 even a very simple data structure like a linked list can be implemented in different ways: pointers among items of the list can refer to the top of the adjacent elements or they can point to a fixed offset within them, data can be stored as part of the items themselves or in auxiliary structures referenced by pointers, and the pointer that signals the end of the linked list can be a `NULL` pointer or a pointer that points to itself. The more complex the structure is, the greater the chances that an OS will implement it by using unique features and tricks, to maximize performance. Furthermore, multiple "basic" data structures can be combined to form more complex ones. For example, hash tables can be implemented as an array of pointers to linked lists and filesystem-related structures often use *tries*, which are a type of search tree whose nodes have a variable number of children. In this case, a node's children could be represented as a linked list, as shown in Figure 5.2.

Since it is impossible to cover all possible data structures and all possible ways these structures *might* be implemented in OSs, we need to establish a hierarchy based on their forensics-relevance. In fact, complex data structures may be extremely rare, may not contain forensic-relevant information, be used only as a fast way to access already organized data, or they might be implemented in a very optimized way which requires deep knowledge of the OS code to be recovered from the dump. At the same time, simple and easy-to-manage data structures could be dominant in fundamental tasks of the OS which do not require leading performance but great stability and easy usage. To guide our selection of data structures that are more commonly traversed during memory forensics, we perform a survey on which of them are required by the Volatility framework [Wal17] to extract information for Linux and Windows memory dumps.

Out of all existing plugins, 79% traverse doubly-linked lists, 23% use arrays, 10% explore trees and 12% extract information from simple linked lists. In the majority of the case (73%) the required information is stored inside the data structure itself or is contained inside auxiliary

C structs pointed by the major data structure. Only 10 of the Volatility plugins require the parsing of more complex data structures, like hash tables, radix trees, and tries. Furthermore, these data structures are used in prevalence to access data that can be also extracted by traversing other data structures, such as the radix tree that Linux uses to connect the process PIDs.

Based on this preliminary study, we decided to focus our techniques on the recovery of single and doubly-linked lists, arrays, and trees. Other data structures could be added in the future, but we believe they can only provide a marginal amount of information during a forensics analysis and can be useful for a more sophisticated analysis that, however, requires a deeper knowledge of the OS internals.

5.3 Terms and Definitions

Before we describe our technique for data structure reconstruction we need to define some terms that we will use over the rest of the chapter.

5.3.1 Atomic Structures

Atomic structures represent the fundamental blocks to store composite information as in-memory ordered collections of fields of different types and sizes. In languages of the C-family, atomic structs are directly mapped to the `struct` data type. In memory, atomic structs are contiguous chunks of data,[1] which contain fields defined in the structure itself along with possible padding inserted by the compiler to optimize the structure size according to the CPU architecture and optimization requirements. Despite having a clear size and a strict separation among its fields at the source code level, in memory an atomic struct appears as a block of bytes without any field separator or any delimiter to separate the structure itself from other adjacent data. While an atomic struct can contain other atomic structs embedded in it, in memory there is no distinction between the fields of the embedded atomic struct and the fields of the embedding one. These ambiguities have important effects on the ability to reconstruct an atomic struct layout starting only from its content in memory.

5.3.2 Data Structures

The OSs use atomic structs and atomic types (such as `int`, `float`, and `struct X *`) as basic blocks to store information by organizing them in data structures.

Linked lists are sequences of atomic structs (the nodes of the list) connected by pointers. Starting from an element it is possible to reach the next one by following a pointer (here generically called next) contained in the element itself. It is important to note that the next pointer can point either to the first address of the next element of the linked list or at a fixed offset inside it. Generally, the last next pointer is a `NULL` pointer or an autopointer (a pointer pointing to itself, see 5.3.3) if the linked list is linear. The linked list may also be *circular*; in that case, the last element of the list points back to the first. Like other

[1] If the system uses virtual memory the atomic struct is a contiguous chunk of data only in the virtual address space, but can be fragmented in physical memory.

structures connected by pointers, the atomic structs of linked lists are not necessarily allocated in contiguous memory locations.

Doubly-linked lists are similar to linked lists. However, elements are linked by two pointers: next and previous (not necessarily located at adjacent offsets), with the latter pointing to the previous element of the list. A doubly-linked list can be seen also as a combination of two linked lists: a linked list composed of the element joined by next pointers and a second list containing the same elements but linked in reverse order through the previous pointers. As for linked lists, doubly-linked lists can be linear or circular. It is important to note that the presence of two pointers linking every element of the list introduces additional information (in the form of a constraint) that helps to recognize doubly-linked lists in memory.

Trees are data structures that may exist in different forms (balanced or unbalanced, RB-trees, etc.). For our purposes, they can be generalized by a generic `n`-ary tree model. In this model, a tree is composed of a root atomic struct linked to a predefined maximum of `n` subtrees, each composed, in turn, by a collection of the same type of atomic structs. At the end of each branch is possible to find leaves that could have a different type of atomic struct.

Arrays are data structures in which atomic types or atomic structs are located in adjacent memory locations. This type of data structure can be very difficult to detect if the atomic type is not a pointer or if the atomic structs contains only non-pointer types because the sequence of elements becomes indistinguishable from an unstructured binary data blob.

Composite data structures provide a way to combine multiple simpler data structures to form more complex ones, such as hash tables and tries.

5.3.3 Pointers

A pointer variable is characterized by two memory addresses: the address of the pointer itself and the address referenced by the pointer. For our goal, we can classify pointers into four categories:

Structural pointers. These pointers represent links between atomic structs that compose a data structure or links among different data structures.

Data pointers. These pointers, contained in atomic structs, reference variables, auxiliary atomic structs which are not part of a data structure but contain information related to it (e.g. `struct pid *thread_pid` in the Linux `task_struct` which references a structure containing the kernel's internal notion of a process identifier), or relations among atomic structs composing the data structure but which do not define its topology (e.g. the `struct task_struct *parent` in Linux `task_struct`).

Autopointers. These pointers have the peculiarity to point to themselves (`*ptr == &ptr`). This property allows to easily identify them inside a memory dump signaling the presence of a field in the atomic struct which, with high probability, has been deputed to point to linked list-like data structures.

`NULL` pointers. `NULL` pointers could be used to signal the end of a linear data structure (as a linked list) or part of a ramified one (a tree). We assume that the OSs and CPU architectures use all-zeros CPU-native words to represent `NULL` pointers. This fact makes a `NULL` pointer virtually indistinguishable from a zero integer variable.

5.3.4 Oracle Function Ω and Γ_x graphs

To identify atomic structs which compose larger data structures it is necessary to retrieve all pointers used by the kernel to maintain relations among single elements. Without any information about the OS and its internals the only way to extract all the pointers from a memory dump is to carve them.

In particular, starting from the CPU architecture is possible to derive the size α of a pointer[2] (e.g. 4 bytes for 32-bit architectures and 8 bytes for 64-bit ones).

We suppose the analyst is able to provide a boolean oracle function Ω (see 5.5 for the implementation we adopted in our evaluation), which takes as input an offset in the dump, the pointer size, and other OS-independent/machine-dependent parameters and returns whether that offset contains or not a possible valid pointer:

$$\Omega(\text{offset}; \alpha, \dots) = \begin{cases} \text{True} & \text{if offset is a valid pointer} \\ \text{False} & \text{otherwise} \end{cases} \tag{5.1}$$

The Ω function allows us to scan the entire memory dump and identify all valid pointers by iterating through overlapping groups of α bytes: the offset in the memory dump (properly interpreted in relation to the physical-to-virtual address mapping) represents the address of the pointer variable while the content of the α bytes located at that offset represent the address of the pointed data. This technique allows to identify both pointers stored at an aligned address (aligned pointers) as well as unaligned ones. It is important to note that we do not specify the concrete implementation of the Ω function: for our technique it is only a prerequisite necessary to extract the pointers, i.e., the analyst has to provide such OS-agnostic function or a function that best approximates its behavior. Furthermore, once the pointers have been extracted, our technique no longer requires information about the hardware configuration of the machine thus also resulting CPU-agnostic.

All pointers that are identified in a memory dump compose a directed graph Γ. In this graph, a pointer (`pointer = &variable`) is represented as an edge connecting the nodes representing the address of the pointer (`&pointer`) to the address pointed by it (`&variable`).

Finally, we define the set of offset graphs $\{\Gamma_x\}_{x \in [X_{min} \dots X_{max}]}$ as the set of graphs obtained by adding a fixed offset of `x` bytes (positive or negative) to all the destinations of the carved pointers (`pointer + x`). For example, $\Gamma = \Gamma_0$, and Γ_{64} is the graph in which each pointer destination is increased by 64 bytes.

5.3.5 Seeds

We define a *seed* as a chunk of data that is either known *a priori* by the analyst or that can be easily recognized as a valid piece of information by inspecting the memory. Seeds can be extracted directly from the memory dump by using OS-agnostic rule-based carving techniques.

[2] α for modern CPUs is also, in general, the “natural" word size and alignment used in memory access.

Examples of seeds are: the name of a process the analyst knows was running when the dump was acquired, a path in the filesystem, the name of a kernel module, an IP address assigned to a network interface, etc. Strings, for instance, can be easily extracted from kernel memory by looking for sequences of ASCII or UTF8/16 characters. The same technique can be used also to extract other information with an immutable and OS-independent structure and encoding such as network packets.

5.4 Approach

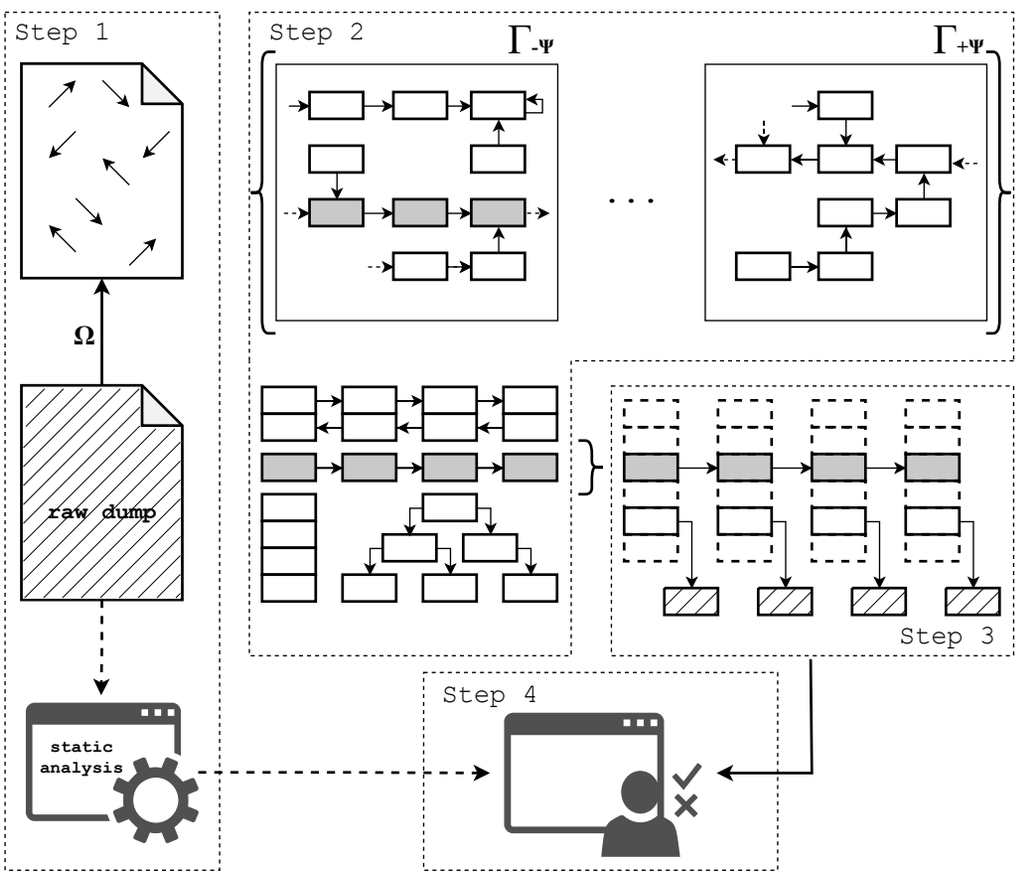


Figure 5.3: OS-agnostic data structures reconstruction phases. Grey cells are pointers of a valid data structure while barred ones contain data.

The goal of our work is to extract forensics-relevant information about the state of the system from a memory dump without any prior knowledge about the OS.

An overview of the process is shown in Figure 5.3. In the first step, we use the provided Ω function to collect the kernel pointers present in the dump. We then locate the seeds by using OS-agnostic data carving and we (optionally) perform a static analysis on the executable pages of the kernel. This analysis extracts cross-references to data structures allocated by the kernel at runtime, which helps our system to prioritize the extracted information and reduce the number of false positives.

After that, we reconstruct relations among pointers (Step 2 in Figure 5.3) by using topological properties of the data structures that we are interested to recognize (discussed in detail in Section 5.4.1). In this phase we reconstruct only the structural skeleton of the data structures: relations among structural pointers do not provide information about the shape (size and limits) of the atomic structs which contain data-pointers and embed forensic-relevant data.

After the skeleton of the data structure is recognized, we perform a statistical analysis of the in-memory raw data that surrounds each structural pointer (Step 3) estimating how many bytes the original atomic data structure extends around them. This allows us to identify also relations among reconstructed data structures and seeds.

Finally, we present to the analyst the identified data structures with the possibility to filter them by looking for specific known seeds or by using cross-references extracted during the static analysis phase (Step 4).

5.4.1 Data Structures Skeleton Recognition

Once pointers are recognized (Step 1 in Figure 5.3), we tackle the task of recognizing the skeleton of data structures. While our approach can be useful to recognize also more complex data structures, as explained in Section 5.2 our current implementation focuses on the most common data structures that are traversed by memory forensics frameworks.

Relation between Lists, Trees and Offset Graphs

Consider the case of lists (both single- and doubly-linked) and trees. These data structures are composed of sequences of atomic structs of the same type linked by structural pointers. Unfortunately, it is quite common to see structural pointers whose destination is not the beginning of the next atomic structure, but rather some point in the middle—e.g., the `next` pointers used in doubly-linked lists in the Linux kernel point to the `next` pointer of the successor's struct [Lov10]. In other cases, e.g., the `prev` pointer in Linux's doubly-linked lists, the destination is at a fixed offset with respect to the same pointer in the adjacent struct (i.e., the `prev` pointers point to the address of `next` pointers of the previous struct).

To generalize, atomic structs part of the same data structure are reached by dereferencing a structural pointer, adding a (positive or negative) offset β to the destination, and repeating the process while keeping the offset β constant. With reference to Linux's doubly-linked lists: $\beta = 0$ for the `next` pointers and $\beta = +\alpha$ for `prev` ones. This process is shown in Figure 5.4 along with the paths it produces in the Γ_β graphs. Hereinafter, we call these paths *chains*.

A chain may correspond to the collection of all the structural pointers of a linked list or parts of other data structures, such as "half" of a doubly-linked list or the branch of a tree obtained by always following a given child link (e.g., all left or right children). However, this procedure generates also false positives, as shown in Figure 5.4. In fact, following a pointer and adding an arbitrary offset x to it could, by chance, lead to another pointer and so on. This produces a valid path on a Γ_x graph that does not correspond to any valid data structure.

A first step to reduce false positives is to note that in Γ_β the atomic structs that belong to true-positives data structures should be at least β bytes large (because otherwise by adding the offset β we would "fall outside" of the data structure). Moreover, atomic structs should also be larger than the pointer size α. Hence, pointers in a valid chain should all be at least at distance

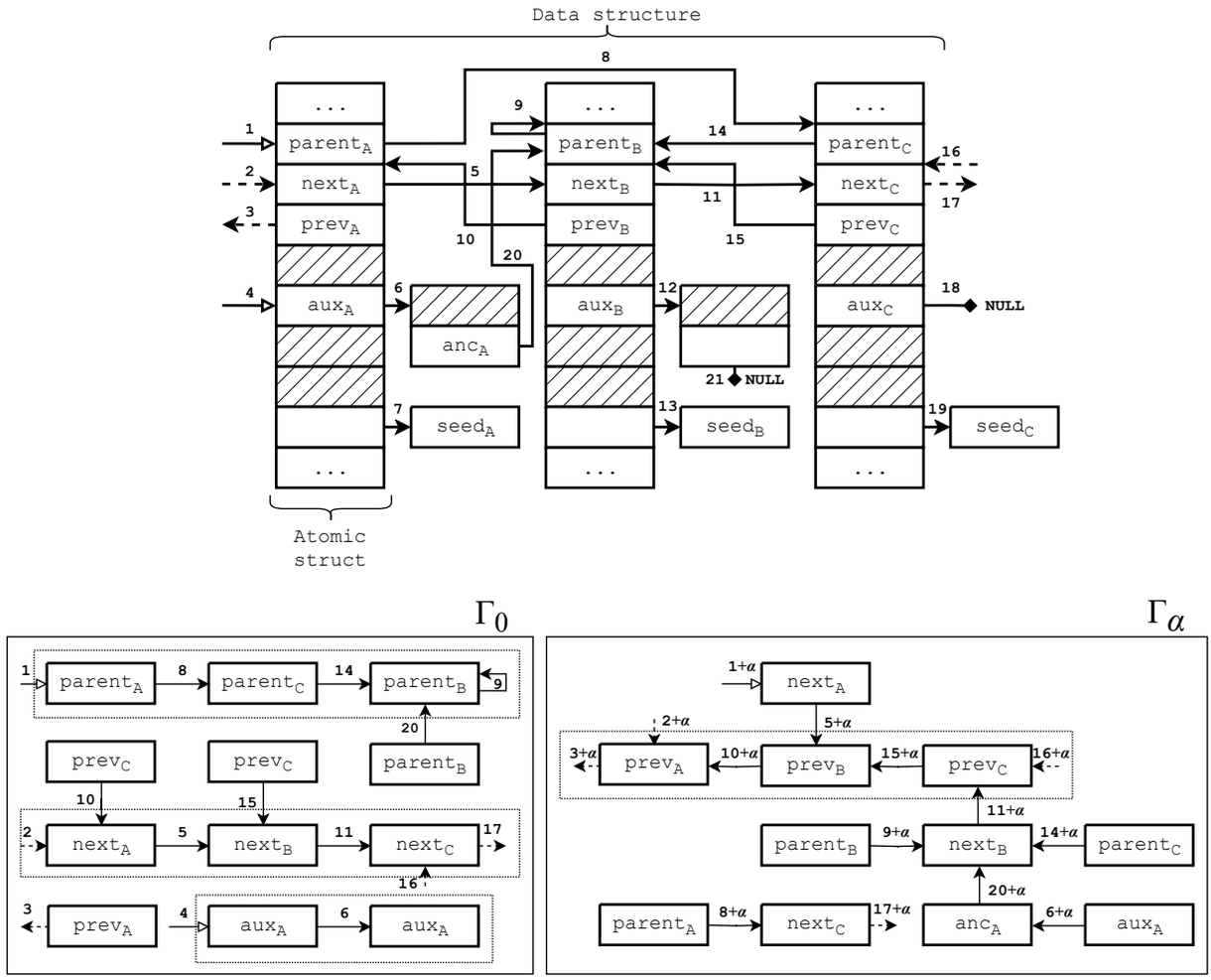


Figure 5.4: Example of data structure and its partial contribution to two Γ graphs. Pointers with empty arrows are false positives due to approximations of the Ω function. Dashed pointers start from structs not represented here. Note pointer 9, an autopointer, and pointer 18 and 21 `NULL` ones. Pointers 6 and 12 are data-pointers (referring to auxiliary atomic structs) as also 7, 13 and 19 (referring to seeds). Cells with diagonal bars contain data. Only contoured chains correspond to meaningful paths and only `next` and `prev` paths involve structural pointers.

$\max(\beta, \alpha + 1)$ from each other. Chains of atomic structs that do not satisfy this requirement can be discarded, as the path from aux_A to prev_A in Γ_α graph in Figure 5.4.

In practice, we compute the offset graphs for a suitable symmetric interval of plausible offsets $[-\Psi, \Psi]$. Ψ is an important parameter that affects the maximum size of atomic structs identifiable by our technique. In fact, in the worse case in which a structural pointer is located at the bottom (last field) of an atomic structure and references the top (first element) of the next element, then the maximum size of the atomic structure that can be retrieved by our approach is Ψ.

It is worth noting that each offset graph has a topological peculiarity: since each memory location contains at most one pointer, the maximum out-degree in the graph is one. In other words, there is at most one outgoing edge from each vertex. On the other hand, several pointers

may have the same destination. This implies that paths in an offset graph may converge; however, there is at most a single outgoing path from any node of the graph. In light of this, chains may contain tail links back to the head forming a cycle, which corresponds to circular lists, but also a "P-shape", where the end of the chain links back to a point in the middle of it. We "cut" chains like these into two separate chains–one circular and one linear (up to the beginning of the cycle).

Doubly-Linked Lists

A doubly-linked list consists of a set of atomic structs with previous and next structural pointers and, like singly-linked lists, may be circular. We allow the destinations of the previous and next links to be at different offsets: for example, next pointers can point to the following next pointer without the need to apply an offset, and previous pointers can point to the preceding next pointer.

We identify doubly-linked lists by recognizing chains that are at a fixed distance and in opposite directions: consider the example of a doubly-linked list containing atomic structs A, B, and C; one chain follows the path $A \to B \to C$, while the other takes the reverse direction $C \to B \to A$.

Trees

Our algorithm recognizes trees by following a bottom-up approach (here for simplicity we illustrate the case of binary trees, however, our approach naturally generalizes to n-ary trees with $n > 2$). We first look for complete trees of depth two by checking the intersection of chains that have a length of two: in this case, they would intersect at a point r, the candidate root, that appears in two different offset graphs, Γ_l, and Γ_r, where we say that offset l reaches the left child and offset r reaches the right one. Then, we perform two checks to verify whether we should discard the candidate tree with root in r: first, we discard all cases in which there is no pointer at offset r from the left child or at offset l from the right child; second, we verify that structs in the candidate tree are distant enough from each other to create a coherent tree (i.e., where the distance respects a lower bound for the size of the struct representing the tree node). We know that this lower bound t should respect the following conditions: (i) $t > 2\alpha$, because the struct needs to contain the pointers to the two children and some payload data; (ii) $t >= \max(|r - l|, |l|, |r|)$, because the destination of the pointers plus the left and right pointers should all fall within the node struct. Hence, we discard all candidate trees for which any two nodes are at a distance that doesn't satisfy this constraint.

Fully balanced trees of arbitrary depth are then constructed recursively by recognizing height-2 trees whose root's children are trees of height $x - 1$ and whose nodes are still at least as distance t from each other as described above. Non-fully balanced trees can be simply obtained by following the l and r links of the trees that we have discovered.

Arrays

As already explained in 5.3.2, this type of data structure can be undetectable if the atomic type is not a pointer and without any knowledge of the array shape or its content. In light of this, we decided to carve only for arrays of pointers, by looking for sequences of in-memory adjacent not `NULL` pointers.

5.4.2 Atomic Structs Size Estimation

The techniques described above allow the reconstruction of relations among structural pointers which maintain links among atomic structs and determine the topology of the data structure. However, to extract forensics-relevant information from atomic structs we first have to infer their boundaries, i.e., the offset at which each atomic structure starts in relation to its structural pointers and its total size. Since we have no information on the OS internals, we can rely only on statistical properties derived from the set of all the atomic structs that compose the data structure. The idea is that fields located at the same offset in the struct contain data of the same type.[3] This phase is important because an under-estimation of the atomic struct size could exclude fields that refer to seeds easily recognizable by the analyst, while an over-estimation could return an avalanche of false positives containing desired seeds complicating the analyst's work.

5.4.3 Computational Challenges

As discussed in Section 5.4.1, to discover doubly-linked lists we look for "parallel" chains running in inverse directions: e.g., if we have a chain connecting structs A, B, C, D, E, we are looking for another chain connecting the same structs in the opposite direction: E, D, C, B, A. We do not have a way to distinguish `next` from `previous` pointer, but for our purposes the two chains are interchangeable—i.e., we care about finding the skeleton of the doubly-linked list, not distinguishing the two directions. Doubly-linked lists are fundamental in many kernels, so we tackled the task of recognizing them with attention, focusing particularly on minimizing the number of false negative cases.

The first issue that complicates our task is that we will not only want to recognize chains whose structural pointers `previous` and `next` have the same destination, but also those that have a fixed offset between them, suggesting that they point to different offsets in the same struct.

A further problem we want to address is that, for a given chain, it is possible that the first x elements do not belong to the doubly-linked list: they may be non-structural pointers from other atomic/data structures that ultimately lead to the doubly-linked lists, we want to recognize while following pointers at the same offset by coincidence as shown in Figure 5.4. Hence, for a given chain we also want to test all its suffixes, i.e., all chains of size 3 or more obtained discarding any number of its first elements.

The two issues described above make it challenging to design a fast algorithm to recognize doubly-linked lists. We solved it by focusing on an invariant that still applies in our case: if we take the *difference* between consecutive pointer locations in two chains composing a valid doubly-linked list, they do not change depending on the offset they reach in an atomic struct. In other words, if `&x` denote the address in memory of a given atomic struct, taking the difference between consecutive pointers in an $A \to B \to C$ chain will return the [(`&B-&A`), (`&C-&B`)] values, irrespectively of the offset in the struct at where the pointers are located.

Crucially, if two chains are indeed the two "halves" of a doubly-linked list, taking the same operation of computing the difference of consecutive pointers on the two halves would return

[3] We are ignoring C-style `unions` which are very difficult to detect without any knowledge of their definition and represent only a small part of the type of atomic struct in kernels (for example, we have measured that in Linux `unions` represents only 4% of the total number of the types of atomic structures definitions).

two lists that have a clear relationship: by inverting one list and changing the sign of all values, we obtain the other. In the $A \to B \to C$ example, for the corresponding $C \to B \to A$ we would indeed obtain the [(`&B-&C`), (`&A-&B`)], which corresponds to the result of inverting and changing all signs of the [(`&B-&A`), (`&C-&B`)] list mentioned before.

Our algorithm is hence based on the above realization: for each chain c containing pointers $p_0, p_1, \ldots, p_n$, we will compute two hashes:

$$h_{c,0} = h([(p_1 - p_0), (p_2 - p_1), \ldots (p_n - p_{n-1})]),$$

which is the hash of sequence of the difference between pointers in it, and

$$h_{c,1} = h([(p_{n-1} - p_n), (p_{n-2} - p_{n-1}), \ldots (p_0 - p_1)]),$$

the hash of the same sequence after inverting it and changing its sign. Since we are interested in the suffixes of c, we will apply the same procedure to all the suffixes of c, i.e., the chains containing the pointers $p_1, \ldots, p_n$, $p_2, \ldots, p_n$ etc. Each pair of chains c_0 and c_1 where $h_{c_0,0} = h_{c_1,1}$ and $h_{c_1,0} = h_{c_0,1}$ is a candidate doubly-linked list.

Using standard hashing functions to compute these hashes is computationally expensive because computing all the hashes for the suffixes of a chain of length n turns out to have $O(n^2)$ computational complexity. In Section 5.4.4 we will discuss our design of a specialized strategy to do that with optimal $O(n)$ computational complexity.

Once the hashes are computed, we now process candidate pairs. We group chains by matching couples of hashes and, out of caution, we verify that we did not have hash collisions (i.e., we verify that all chains have the same length and are "parallel": corresponding structs have a constant distance between them).

We want to assign each chain and each pointer to at most one doubly-linked list: any candidate chain that will be processed having a pointer already assigned to another doubly-linked list is immediately discarded. The longest matching chains are those that have both the lowest probability of being false positives and the most informative.

Once we have a candidate chain $c = [p_0, p_1, \ldots p_{n-1}]$, we consider the smallest distance between its elements ordered by their addresses. These pointers should be at the same offset of different atomic structs, so the smallest value of the differences $t = \min_{i \in 1 \ldots n}(c'_i - c'_{i-1})$ is a higher bound for the size of the atomic structs pointed by c. Among the chains that are candidate matches to c, we take the one at the smallest offset o from it. If $o + \alpha \leq t$, then we consider the two chains to be a successful match and output them as a doubly-linked list.

5.4.4 Bi-Directional Hashes

In the first iteration of our implementation, we found that the part of the hash computation was a severe bottleneck because for each list of length n we had to compute 2 hashes for each of its suffixes, for a $O(n^2)$ computation complexity for each chain of length n that turned out to be untreatable because we often encountered long chains, leading to expensive computations.

We can state our problem as that of computing, for a list $l = [v_0, \ldots, v_{n-1}]$ having n elements, all the values such $h_{0,i}$ and $h_{1,i}$ such that

$$\begin{cases} h_{0,i} = h([v_i, v_{i+1}, \ldots, v_{n-1}]) \\ h_{1,i} = h([-v_{n-1}, -v_{n-2}, \ldots, -v_i] \end{cases}$$

for all values of $i \in \{0, 1, \ldots, n-1\}$, where h is a suitable (non-cryptographic) hashing function.

We define our hash function to be

$$h([x_0, \ldots, x_{n-1}]) = \sum_{i=0}^{n-1} a^i x_i \mod 2^{64}, \tag{5.2}$$

where a is a constant.[4] This function is inspired by rolling hash functions such as those used in the Rabin-Karp algorithm [KR87]. The modulo 2^{64} arithmetic is implemented simply and efficiently by using unsigned 64-bit integers (and ignoring overflows).

The $h_{0,i}$ values are computed simply by noting that

$$\begin{cases} h_{0,n-1} = v_{n-1} & \mod 2^{64} \\ h_{0,i} = v_i + a \cdot h_{0,i+1} & \mod 2^{64} \quad \forall i \in \{n-2, \ldots, 0\}. \end{cases} \tag{5.3}$$

Hence, we can compute all these values simply by starting with the known value of $h_{0,n-1}$ and looping back to $h_{0,0}$ using 5.3.

For the $h_{1,i}$ values, we need to find the multiplicative inverse of a, that is $a^{-1} \mod 2^{64}$. We can do that using the extended Euclidean algorithm [Knu98].[5] Similarly to 5.3, we compute auxiliary values t_i such that

$$\begin{cases} t_{n-1} = -v_{n-1} & \mod 2^{64} \\ t_i = -v_i + a^{-1} \cdot t_{i+1} & \mod 2^{64} \quad \forall i \in \{n-2, \ldots, 0\}. \end{cases}$$

We finally compute

$$h_{1,i} = a^{n-1-i} t_i \mod 2^{64} \quad \forall i \in \{0, \ldots, n-1\}.$$

It is not hard to verify that values computed in this way satisfy 5.2.

In a further optimization, we recognize that chains can be "confluent", i.e., they have a common suffix. When this happens, we do not recompute the hashes already computed for suffixes already examined, and we reuse the values $h_{0,j}$ and t_j already computed for the common suffix of the chains.

5.5 Implementation

In this section, we explain how we implemented the data structures recognition techniques in our proof-of-concept tool, Fossil, how we reduced the number of false positives by introducing OS-agnostic heuristics, and the dataset used for experiments.

5.5.1 Ω Function

The role of the Ω function is to identify values that can be valid kernel pointers in the memory dump. Unfortunately, without any additional information about the OS, this is a very difficult task, as potentially any number could be a valid kernel address. Luckily, the system

[4] In our implementation, $a = 6364136223846793005$ [Knu98].

[5] In our implementation, $a^{-1} = 13877824140714322085$.

architecture (which we assume to be known) can provide useful information. For instance, on micro-controllers with a 32-bit memory address range, MMIO, and no virtual memory (e.g. PIC32, or Renesas RX) it is possible to drastically reduce the values of valid memory addresses and build a very precise Ω function. On CPU architectures with a memory protection unit (MPU), such as ARM Cortex-M, it is possible to infer which regions of the RAM are writable and could contain valid pointers.

As explained in Chapter 4 we developed an OS-agnostic technique based on the functioning of MMUs, which is able to automatically retrieve the kernel virtual address space from the memory dump for Intel x86/AMD64, ARM/AArch64 and RISC-V 32/64-bits CPU.[6] This allows us to define an approximated oracle function that marks all valid virtual addresses of the kernel address space as potential pointers by scanning the kernel pages contained in the memory dump looking for sequences of α bytes which can be interpreted as valid virtual memory address belonging to the virtual address space of the kernel. This, however, does not guarantee that the discovered sequences of α bytes correspond to real pointers: a group of continuous α bytes that, accidentally, can be interpreted as a valid kernel virtual address, produces a false positive. However, this approach has no false negatives because structural pointers belonging to kernel data structures must reside in the kernel virtual address space to be reachable and accessible by the kernel itself. It is also possible to use other metadata available in the kernel radix-tree's page tables entries to further restrict and refine the Ω function. For instance, we limit our analysis to those memory regions exclusively accessible by the kernel and denied to user space programs because fundamental data structures maintained by the kernel must be protected from the possible corruption of a malicious user space process. While the extraction of the kernel address space depends on the architecture, as shown in Chapter 4, our technique does not and it is completely CPU-independent.

5.5.2 Dataset

To test the accuracy and performance of our OS-agnostic data structure recognition technique we assembled a dataset of memory dumps taken from virtual machines (VMs) running 14 different OSs. The majority of these systems are running on x86/AMD64 (due to the abundance of the available OSs for this architecture) but we also included two OSs running on AArch64: a Linux machine and an iPhone running iOS. We specifically included the Linux Debian machine for both AMD64 and AArch64 architectures to compare the results and check whether our technique introduces some bias, and iOS as an example of a real and complex OS target that is not supported by other forensic tools. The experiments are conducted on virtual machines to easily collect atomic snapshots of the physical memory and avoid inconsistencies [PFB19]. Excepting for the iOS VM, which requires 6 GB to run, each VM was configured with 4GB of RAM because the x86 32-bit architecture cannot address more memory and, at the same time, this is the typical memory size of many IoT devices.

The fourteen OSs, shown in Table 5.1, include general-purpose, embedded, mobile and amateur projects and adopt different designs and architectures (including hybrids, monolithic, real-time, and micro-kernels). For our tests, we decide to focus on OSs written in C or C++ due to their overwhelming prevalence and to the 1-to-1 mapping of atomic data structures to the `struct` construct present in these languages. There have been some attempts to write

[6] We have also shown that, for some CPU architectures, a full OS-agnostic reconstruction of the kernel virtual address space is not possible (PowerPC) or requires a manual effort from the analyst (MIPS).

prototype OSs in other languages, such as Rust or Haskell. However, these languages envelope atomic data structure into more complex constructs for either security (Rust) or abstraction (Haskell), requiring a preliminary understanding of the language constructs to perform atomic data structure mappings.

5.5.3 Static Code Analysis

To extract global references to kernel data structures, we perform a static analysis of the executable pages located in the kernel virtual address space. Using MMUShell 4 we reconstruct the kernel address space and virtual pages' permissions from the memory dump and save them as an ELF core file. This file contains kernel pages mapped as ELF segments with access permissions equal to the permissions set by the OS in the page table entry associated with the mapped data page. Then we analyze the ELF with a Ghidra [Nat23] plugin, which disassembles the instructions and performs a simple static analysis to identify the address of all the global variables referenced in the code.

5.5.4 False Positives Reduction

As discussed in Section 5.4.1, chains on Γ graphs are directly related to the skeleton of different data structures. However, as shown in Figure 5.4, a large number of links are in fact false positives and therefore we employ some heuristics to clean the extracted data.

To begin with, we define a hierarchy among data structures in relation to how much strict their topological constraints are since it is less probable that false positives are generated if topological constraints are strict. Doubly-linked lists have the strictest constraints requiring a fixed distance between the structural pointer which composes the `next` and `prev` chains as already explained in Section 5.4.1. Binary trees have looser constraints in comparison with doubly-linked lists as they only require that each node has two valid (and not both `NULL`) pointers at a fixed distance from the address pointed by the parent node. Looser constraints are required for arrays, for which we only require more than two consecutive pointers. At the lower end of the scale we have linked lists, for which there is only the requirement to represent paths in Γ subgraphs, and sets of auxiliary atomic structs which are obtained by following data pointers at a fixed offset of already discovered data structures. With this hierarchy in mind, we assume that structural pointers that our technique associates with a data structure with tighter constraints cannot be a structural pointer of looser constrained data structure (in other words, we assign each structural pointer to only one, the most constrained, data structure).

We then proceed to filter each type of data structure individually. The heuristics we use are often based on the locality of data and fields (e.g., the fact that right and left pointers in a tree are logically placed closer to each other in the struct definition) and use thresholds that can be configured and fine-tuned by the analysts (e.g. the minimum number of atomic structs in lists and doubly linked lists to be considered valid candidates).

Doubly-linked lists

As explained in Section 5.4.1 linear and circular doubly-linked lists can be seen as a couple of chains belonging, in general, to two different Γ offsets graphs. Supposing that the implementation of the doubly-linked list used by the kernel is unique we consider as valid linear and

circular doubly-linked lists those composed of chains with the most abundant pair of offsets among the doubly-linked lists reconstructed by our tool, filtering out possible false positives data structures based on chains that accidentally match.

Trees

We assume that the `left`, `right` and (optionally) the `parent` pointers are close inside the node structure. Thus, our tool looks for binary trees made of chains obtained from structural pointers extracted from Γ graphs with offsets in $[-8\alpha, 8\alpha]$ and composed of at least 2 levels (7 nodes, including the root one).

Arrays

We consider an array to be valid only if it contains at least three non-`NULL` consecutive elements, not already used for previous data structures. Moreover, as arrays are abundant as part of other structures, we only report those that are directly referenced by the kernel code (e.g., through global variables) as identified by our static analysis.

Linked Lists and Sets of Auxiliary Atomic Structs

These two classes of data structures are the most prone to generating false positives. In fact, every pointer can be a valid starting point of a linked list and every field containing pointers of an already discovered data structure can be the root of a set of auxiliary structs. Therefore, for linked lists we apply the same heuristic we used for arrays, limiting the report to those structures that are referenced in kernel code. Furthermore, assuming that the implementation of doubly-linked lists are based on the single linked lists one, we consider only linked lists generated starting from Γ graphs with offsets belonging to the most abundant couple of offsets used to generate doubly-linked lists, and of length greater than 2. For sets of auxiliary atomic structs, we consider only fields of atomic struct of higher level which contains at least 90% of not `NULL`.

5.5.5 Size of Atomic Structs

To estimate the boundaries of each atomic data structure we analyze the area of memory surrounding the pointers identified in the previous steps. First of all, we assume that an atomic struct can contain a maximum of 1024 fields of size α. This results in data structures with a maximum size Ψ of 4KB for a 32-bit architecture and 8KB for 64-bit. This parameter is necessary because otherwise the entire memory could be considered as a single atomic struct.

We then estimate the size of the atomic struct which belongs to the same data structure as:
$\Psi_{\text{ex}} = min(\Psi, mode(|p - q|))\forall p, q \in \{\text{structural pointers}\}$
In other words, we consider the minimum between the chosen maximum Ψ value and the typical distance among the structural pointers. This estimation is based on the assumption that allocations of atomic data structures are not isolated events and/or the kernel uses an optimized memory allocator (as SLAB system in Linux kernel). If this is true, at least two atomic structs of the same data structure are allocated contiguously in memory, so the distance among their structural pointers is an upper limit for the size of the atomic struct itself. However, instead of considering the minimum distance among all the structural pointers as an estimation

of the size, we consider the more conservative limit given by the typical distance among them. This allows for cases in which one of the atomic structs has a smaller size (as is the case for the first element of various doubly-linked lists in the Linux kernel).

After the size of the atomic struct has been estimated, we have to determine its alignment. In fact, we have an estimation for the atomic struct size but we don't know yet what is the offset of structural pointers inside the atomic struct or, in other words, where the atomic struct begins. Before proceeding to the alignment estimation, we have to determine which offsets (aligned or not) contain valid pointers in the range of $[-\Psi_{ex}, \Psi_{ex}]$ around structural pointers (the intervals contain all possible positions of the Ψ_{ex} window). We consider an offset as containing pointers if at least the 90% of the atomic structs of the data structure contain a valid pointer or a `NULL` element at that offset. We use a threshold because, as already said, some atomic structs can have a different shape due to their functions (the head of a list or a root of a tree), and because the Ω function can introduce false positives (data bytes interpreted as pointers). Last but not least, this threshold also partially mitigates the uncertainty introduced by pointers contained in `union` fields.

To estimate the correct alignment we rely on two heuristics. The first one is based on the presence of data pointers that maintain auxiliary links among structs. An example of it is the `struct task_struct *parent` in Linux `task_struct`, which points to the start of the atomic struct of the parent process. Checking if at a fixed offset at least 90% of the pointers refer to addresses which distance the same quantity from one of the structural pointers allows to identify them. If all these pointers reference the first byte of another atomic struct of the same data structure we can determine the exact alignment of the structure in the Ψ_{ex} window. The second heuristic is used if the first fails and it considers the minimum and the maximum offset which contain valid not-`NULL` pointers, for at least 90% of the atomic structs compatibly with the atomic struct size already estimated.

5.6 Data Structure Recovery

Table 5.1 reports, for each OS, statistics about the pointers recovered by the Ω function used in our experiments, highlighting noteworthy values.

The number of pointers discovered by the Ω function largely depends on the size of the pointer type (α) and the main language used to write the OS. Columns 6 and 7 of Table 5.1 show that for 32-bit OSs, our approach returns the highest number of pointers relative to the size of the kernel address space. This abundance is due to the high number of data bytes whose value falls into the kernel address space and thus can be confused with real pointers. On the other hand, the Intel 64-bit architecture (as well as AArch64) requires kernel addresses to have the higher bits set to 1, which introduces a strong constraint on which 8-ple of bytes could represent valid addresses.

Furthermore, the programming language also affects the number of pointers present in the kernel address space. As shown in Table 5.1 and highlighted in grey, the OS with the highest number of recovered pointers is HaikuOS 32-bit – which is written mainly in C++. The C++ memory layout is known [SB03] for introducing overhead due to virtual dispatching, virtual inheritance, and dynamic typing, resulting in a large number of pointers.

We note also that some OSs make extensive use of autopointers (which are almost false positive free due to their definition) while others do not use them at all. A mere glance at

Table 5.1: OSs dataset and recovered pointers.

OS					Pointers		
	Kernel type[1]	Open-source	Main language	Pointers size (α)	Total	Pointers over total locations ratio%	Autopointers over pointers ratio %
Darwin	H	●	C	8	9.7×10^5	< 0.1	3.3
Embox	R	●	C	4	3.9×10^5	0.15	< 0.1
FreeBSD	M	●	C	8	2.2×10^6	< 0.1	< 0.1
HaikuOS	H	●	C++	4	1.76×10^7	8.99	< 0.1
HelenOS	m	●	C	8	4.4×10^5	< 0.1	6.2
iOS (AArch64)	H	●	C	8	1.5×10^6	< 0.1	0.8
Linux	M	●	C	8	5.6×10^6	0.12	20.99
Linux (AArch64)	M	●	C	8	5.5×10^6	0.25	21.39
NetBSD	M	●	C	4	5.1×10^6	5.33	< 0.1
ReactOS	m	●	C	4	3.5×10^6	1.47	0.3
ToaruOS	H	●	C	8	6.7×10^5	< 0.1	0
vxWorks	R	○	C	8	2.1×10^5	< 0.1	0.8
Windows XP	H	○	C	8	9.9×10^5	0.27	16.89
Windows 10	H	○	C	8	1.9×10^6	0.15	6.3

[1] H: hybrid-, m: micro-, M: monolithic-, R: real-time kernel

the percentage of autopointers in the recovered pointers can tell something about the kernel internals: usually, autopointers are used to represent (doubly) linked lists with zero elements or signal their ends. This is confirmed by Table 5.1: OSs like Linux and Windows which are known to use autopointers for zero-length linked lists show an abundance of them. From the point of view of an OS-agnostic memory forensics approach, the presence of a large number of autopointers is an important piece of information. For instance, it allows the analyst to infer that fields at the same offset in atomic structs which contain autopointers are "roots" of auxiliary linked lists and, in general, it allows to recover information on how the linked lists are implemented in the OS without analyzing the kernel code delegated to their management.

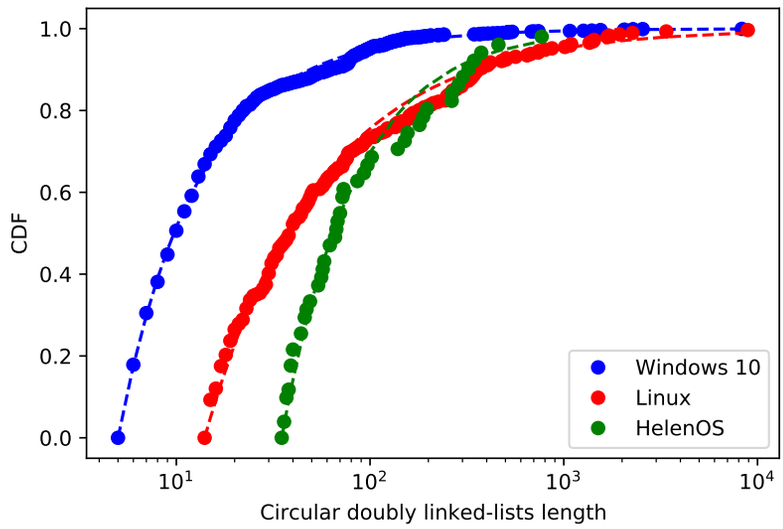
Finally, as expected for the same kernel running on different CPU architectures, we note that the chosen Ω function extracts the same number of pointers for AMD64 and AArch64 Linux with, approximately, the same ratio of autopointers.

Table 5.2 summarizes the number of data structures identified by our tool for each OS. In general, arrays of pointers (to strings or other structs) are the more abundant. In a few cases, our system was not able to find any results for some particular type of data structures. This can be due to a real absence of that data structure (for example, because all doubly-linked lists for a given OS are circular and the tool was able to reconstruct them without producing false positive linear ones) or can be a consequence of a more complex implementation that is not currently recognized by our tool (for example, trees in HaikuOS). As for pointers, the number of data structures extracted for AMD64 and AArch64 Linux is approximately the same, confirming the CPU architecture independence of our technique which does not introduce biases.

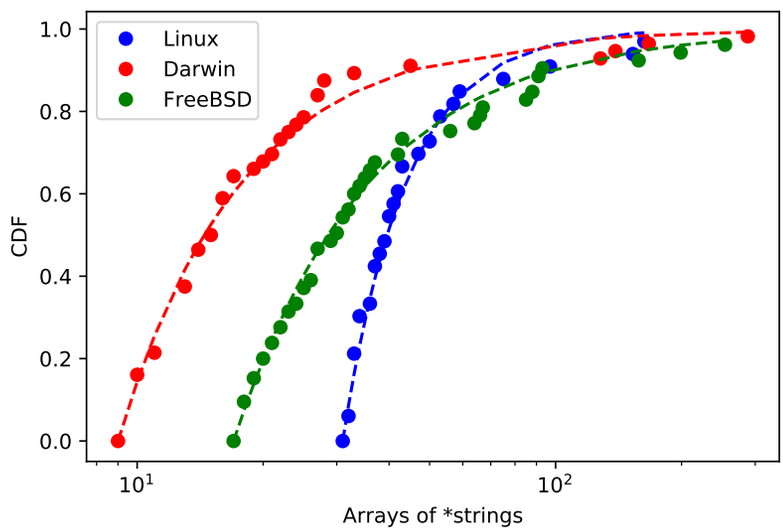
Table 5.2: Data structures (data structures) recovered.

OS	Linear D.-L. L.	Circular D.-L. L.	Trees	Arrays of *structs	Arrays of *strings	Linked Lists
Darwin	11	385	127	1214	1801	35
Embox	0	22	35	1131	795	6
FreeBSD	86	0	993	1008	895	41
HaikuOS	4117	64	0	305	232	1184
HelenOS	25	1173	127	41	45	1
iOS	20	256	192	5234	229	36
Linux	120	3632	1034	693	5947	46
Linux (Aarch64)	110	3362	936	229	4985	43
NetBSD	41	18	1218	1482	406	45
ReactOS	7	200	49	492	325	12
ToaruOS	101	0	14	62	229	15
vxWorks	51	14	199	349	416	13
Windows XP	38	889	228	463	206	20
Windows 10	145	6639	36	0	282	0

In Figure 5.5a we have plotted the cumulative distribution function of the length of circular doubly-linked lists for the three different OSs in which they are more abundant. Dashed lines are power-law fits obtained by using the technique developed by *Clauset et al.* in [CSN09] and tested against an exponential distribution using the loglikelihood ratio ($P<0.05$). While all the three OSs have circular doubly-linked lists of different magnitude of orders in length, it is possible to note some differences among the distributions. For example, 80% of circular doubly-linked lists in Windows 10 contain less than 11 elements. To reach the same percentage in Linux, lists contain instead up to 200 elements. This fact can suggest to an analyst that the OS under analysis, such as Linux in this case, could make extensive use of this type of data structure also to store small pieces of information. A similar analysis can be made by looking at Figure 5.5b, which shows the cumulative distribution function of the size of arrays of pointers to strings. In this case, instead, all the OSs prefer to use small arrays of pointers to strings instead of a longer one, which often contains debug strings inserted during the kernel compilation, as in the case of the three OSs considered. The power-law trend of cumulative distributions in cycles and arrays suggests also that is impossible to define simple heuristics to filter out false positives based on a cut-off on the length of cycles/arrays derived from their distribution in the dump. We have also analyzed the distribution of the depth of recovered trees in three OSs (NetBSD, Linux and FreeBSD) chosen with the same criteria as above, however, in this case, the distributions do not show the power-law behavior and have a maximum depth for recovered tree equal to 8 (equivalent to a maximum of 256 nodes).



(a)



(b)

Figure 5.5: Circles: cumulative distribution function of the length of circular doubly-linked lists (a) and arrays of pointers to strings (b) for the three OSs in which are more abundant. Dashed lines: power-law fit of the distributions.

5.7 Seed-Based Data Structure Identification

We start our analysis with the most common example of memory forensics: process enumeration.

The set of running processes is fundamental to understand the system status at dump time, because it allows the analyst to determine if there were malicious or anomalous processes running on the system, making this the first target, both in terms of priority and relevance,

Table 5.3: False positives vs seeds.

OS	Process List		
	Structure type[1]	Two seeds all/referenced	One seed all/referenced
Darwin	LDL	6/6	9/9
Embox	AS	5/4	8/5
FreeBSD	L	4/4	34/23
HaikuOS	LDL	3/3	114/114
HelenOS	CDL	2/-	5/-
iOS	LDL	3/3	9/7
Linux	CDL	3/1	76/3
Linux (AArch64)	CDL	1/1	300/2
NetBSD	LDL	2/2	2/2
ReactOS	CDL	5/4	5/4
ToaruOS	A1	1/1	47/47
vxWorks	LDL	2/-	14/-
Windows XP	CDL	3/3	5/3
Windows 10	CDL	4/-	6/-

[1] AS: array of pointers to structs, A1: first level auxiliary structure, CDL: circular doubly-linked list, LDL: linear doubly-linked list, L: linked list

of most memory analyses. To test our technique we simulated the scenario, summarized in Table 5.3, in which the analyst knows either one or two processes' names (the seeds) and wants to retrieve the names of all the other processes running in the system and the data structures that contain or point to them. However, the analyst does not know anything about the OS itself, nor the type of data structure the OS uses to maintain processes information. To identify possible seeds we carved the kernel virtual address space for ASCII and UTF-16 strings composed of more than 2 characters looking also for strings in read-only pages because some forensics-relevant strings can be static and not created at runtime (for example kernel threads' names).

The second column of Table 5.3 reports the data structure type of the true process list *as identified by the tool*. It is important to note that, in general, the type identified by the tool can correspond only partially to the exact data structure type of the process list: the presence of false positive pointers, due to the use of an approximated Ω function, can affect, for example, the reconstruction of doubly-linked lists which will appear as two separated linked lists as discussed in Section 5.3.2. Most OSs use either single or doubly linked lists to maintain the process list, directly embedding the process name or a pointer to it in the atomic structs. We found two exceptions to this rule. The first is Embox which keeps track of processes through an array of pointers to atomic structs, which in turn contain the process metadata. The second is ToaruOS, which uses a 'light' linked list with atomic structs containing only a pointer to an auxiliary atomic struct that actually contains the process metadata and its name.

Our tool was always able to automatically identify the correct data structure and print all the processes' names. However, the tool often identified few candidate structures that pointed to the seeds processes. If the two seeds were uncommon strings, such as `sctp_iterator` for

HaikuOS or the well-known `swapper/0` for Linux, the number of data structures identified by our tool as a possible list of processes was always either one or two at the most. If instead, the seeds were more common strings that could appear also in other structures, the tool emitted more possible options. To check this eventuality, in the third column of Table 5.3, we have reported the *worst* case obtained by testing the reconstruction technique with the two process name seeds that are the most referenced by data structures. Each cell of the column contains two values: the first one is the total number of data structures referring to the two seeds, while the second one is the part of them that is referenced by the kernel code as deduced by static analysis. For two OSs (HelenOS and vxWorks) Ghidra was not able to extract a correct reference to the true process list while for Windows 10 Ghidra was not able to manage the high number of segments that compose the kernel ELF core file and consequently extract references. The table shows that for all the OSs our tool was able to identify the process lists also in the worst-case scenario, outputting at most six possible options for the analyst to check.

In the case in which the analysis knows only one process name and wants to retrieve all the other ones, the fourth column of Table 5.3 reports again the *worst* case scenario. For most of the OSs the set of possible structures remained low, with the exceptions of HaikuOS and ToaruOS, which generated respectively 114 and 47 options. In the case of HaikuOS, the language used, as discussed in Section 5.6, produces a high number of pointers which in turn generate a lot of false-positive chains in Γ graphs. While for ToaruOS, we have counted structures and first-level auxiliary structures reachable by them, which has increased the false positive rate, since the true process list is in an auxiliary struct and an analyst does not know it a priori. In the case of one seed, the references extracted using static analysis can be more useful, significantly reducing the number of options in the cases of FreeBSD and Linux.

Finally, we investigated whether the extracted data structures could also be used to retrieve other non-textual information relative to each process. For example, for all cases that use single/doubly-linked lists it was possible to easily infer the process ID (PID) of each process by querying for integer fields in ascending order (since newer processes are always appended at the end of the list). Also, the same technique can be used to extract timestamps (e.g., related to the starting time of each process) and, having an estimation of the boot time in the case the OS uses timestamps relative to it, the same technique allows also to recover the time at which the process is started.

5.7.1 Other Forensics-relevant Data Structures

During a forensics analysis of an unknown OS, there are other data structures that can be useful as a starting point for more advanced analysis of the system. For instance, the set of loaded kernel modules, the set of the kernel memory pools, and information about the mounted filesystems. These data structures can help the analyst to understand the internals of the OS and to extract information that even alone can suggest a compromise of the system. The data structure referencing kernel modules, for example, contains information about modules loaded at dump time, making it possible to better understand how the kernel interacts with hardware and identify possible malicious modules loaded by an attacker. The kernel memory pools, such as Windows pool system or SLABs in Linux, are, instead, used by the kernel to allocate memory dedicated to contain other kernel objects divided by their type or size. The identification of pools allows the analyst to deeply explore the memory space of the kernel, understanding how it is managed and what it contains. They are so fundamental that some

forensics tools, such as Volatility [Wal17], base their complete analysis of Windows dump on the ability to identify kernel pools. Finally, the kernel manages real or virtual filesystems to organize data on second storage memories or to expose features or statistics, such as the `/proc` or `/sys` virtual filesystems in Linux. Understanding which filesystem is active at dump time can facilitate disk forensics and increase the knowledge about kernel interface (through virtual filesystems) which may have been abused by malicious user space programs.

For each OS in our dataset, we have collected two kernel module names, two kernel pool names and various strings relative to filesystems (such as filesystem names or the device name associated with hard drives by the kernel) and then used our tool to automatically identify data structures containing them and all the other elements of the same type. Filled circles "●" in Table 5.4 show for which OS our tool was able to successfully extract this information, while those marked with the "†" superscript (e.g., Linux kernel modules) means that the tool found two different data structures that correctly referenced the two seeds. This is very important because sometimes malicious software tampers with kernel data structures to hide their presence but, if we know more data structures are used to link the same atomic structs, it is possible to detect differences among them that might be a sign of infection. For some OSs (Embox, FreeBSD and ToaruOS) it was not possible to detect any data structure related to kernel pools because these OSs do not have them. In other cases, our technique fails to detect some of the data structures because kernels use more complex ones which are not detectable by our proof-of-concept tool or due to multiple levels of references to auxiliary structures. An example of the first case is the kernel modules "list" of ToaruOS which is implemented as a hash table, detected by our tool only because, internally, it is implemented using a linked list.

In the fifth column of Table 5.4 we reported other data structures our tool was able to correctly identify by using two known seeds. It is interesting to note that it is possible to obtain information about the interaction between the process and the kernel (communications channels as sockets, pipes, locks and semaphores), information about the hardware (the list of network cards and devices detected), additional information on the kernel itself (kernel parameters, kernel internal tasks) or relative to the user space programs (associated linked libraries and environment variables). All this information is extremely useful to an analyst who has to face an unknown OS to speed up the forensics process.

These results show the main use of our tool: the analyst extract strings or other known datatypes from memory, then pick a few entries and use our tool to test whether any data structure exists that points to the two seeds. If so, the tool also output all the other elements of the data structure and prints the corresponding elements that complement the seeds' values.

5.8 Comparison with Other Tools

It would be interesting to measure how many data structures identified by our system are *'correct'*, i.e., they correspond to real structures created by the operating system, and how many data structures we miss. However, this test requires access to ground-truth information regarding the exact data structures present in memory at dump time, also including those unreachable and already de-allocated, since they can still be relevant for forensics investigations. Unfortunately, we are not aware of any tool or technique that can provide this information, even in the case of open-source OSs. Thus, we propose to evaluate the effectiveness of our solution by comparing the forensics-relevant data structure recovered by our approach with

Table 5.4: Other forensics-relevant data structures.

OS	Kernel modules	Kernel pools	File systems	Other structures
Darwin	●	●	●	• List of network devices • System locks • Kernel/user pipes • Kernel parameters
Embox	●	-[1]	○	• List of commands
FreeBSD	●	-[1]	●	
HaikuOS	●	●	○	• Executable libraries • Kernel/user pipes • Semaphores
HelenOS	●[†]	●	●[†]	
iOS	○	●	●	• List of network devices • System locks • Kernel/user pipes • Kernel parameters
Linux	●[†]	●	●	• Files in `sysfs` • Network protocols
Linux (AArch64)	●[†]	●	●	• Files in `sysfs` • Network protocols
NetBSD	●	●[†]	●	• Kernel tasks
ReactOS	○	●	○	
ToaruOS	●	-[1]	●	• Devices' list • Processes' environment
vxWorks	○	●	○	• Devices' list • Open sockets
Windows XP	●	●	○	
Windows 10	●	●	●	

[1] Not supported by the OS.
[†] Two different types of data structures correctly reference the same seed.

those extracted by other state-of-the-art tools.

For instance, rule-based systems, such as Volatility, are able to recover forensic-relevant data structures by using custom hand-written rules, specifically tailored for a given version of an operating system. Therefore, for each Volatility Linux plugin, we manually extract the list of data structures used as starting points to extract information from the memory dump. We then check whether Fossil is able to find those same structures in the Linux Debian x86 memory dump used in our experiments.

For this comparison, we analyzed 56 Volatility plugins – which correspond to all available commands after removing those that do not involve data structures or that operate on user space. Out of these, four relied on complex data structures that are not supported by our prototype (i.e., bit-vectors, hashtables and n-ry trees).

Of the final 51, Fossil was able to correctly recover all data structures used by 69% of the Volatility plugins. Starting from those data structures the plugins perform various types of analysis: some of them directly extract the required information, while others access connected auxiliary atomic structs. In all these cases, if the information was in textual form, Fossil reported it by default in its output. Other plugins start instead a graph exploration de-referencing pointers at various offsets, no longer representing coherent, and therefore identifiable data structures, but only paths in the set of Γ graphs. In this case, Fossil correctly identified the data structures used as starting points but was not able to automatically extract the final piece of information. Detailed results are reported in Table 5.5.

For the remaining 31% of the plugins, the heuristics we use to limit the time complexity of our prototype implementation prevented our tool from recovering the required data structures.

Table 5.5: Fossil's ability to detect data structures used by Volatility plugins.

Plugin	Seekable by Fossil	Found by Fossil	Plugin	Seekable by Fossil	Found by Fossil
apihooks	●	●	list_raw[8]	●	○
arp[1]	●	○	lsmod	●	●
aslr_shift[2]	○	○	lsof[7]	○	○
banner[2]	○	○	malfind	●	●
bash[3]	○	○	memmap	●	●
bash_env[3]	○	○	moddump	●	●
bash_hash[3]	○	○	mount	●	●
check_afinfo[1]	●	○	mount_cache[1,6]	●	◐
check_creds	●	●	netfilter[8]	●	○
check_evt_arm[4]	○	○	netscan[2]	○	○
check_fop[6]	●	●	netstat	●	●
check_idt[2]	○	○	pidhashtable[7]	○	○
check_inline_kernel[2]	○	○	pkt_queues	●	●
check_modules[1]	●	○	plthook	●	●
check_syscall	●	●	proc_maps	●	○
check_syscall_arm[4]	○	○	proc_maps_rb	●	●
check_tty	●	●	procdump	●	●
cpuinfo[2]	○	○	process_hollow[3]	○	○
dentry_cache[6]	●	●	psaux	●	●
dmesg[5]	○	○	psenv	●	●
dump_map	●	●	pslist	●	●
dynamic_env[3]	○	○	pslist_cache[6]	●	●
elfs	●	●	psscan[2]	○	○
enumerate_files[6]	●	○	pstree	●	●
find_file[7]	○	○	psxview[6]	●	◐
getcwd	●	●	recover_filesystem[6]	●	○
hidden_modules	●	●	route_cache[6]	●	○
ifconfig[8]	●	○	sk_buff_cache[6]	●	●
info_regs	●	●	slabinfo	●	●
iomem[7]	○	○	strings[8]	●	○
kernel_opened_files[8]	●	○	threads	●	●
keyboard_notifiers[8]	●	○	tmpfs	●	●
ldrmodules	●	●	truecrypt[8]	●	○
library_list	●	●	vma_cache[6]	●	●
librarydump	●	●	yarascan[2]	●	●

– Highlighted rows: Fossil can identify data structures that should be used by the plugin but which are missed by Volatility.
– ◐: Fossil identifies only part of the needed data structures.
[1] Less than 90% of pointed/embedded strings in the data structure.
[2] Plugin looks for unstructured data.
[3] Data structure in user space.
[4] Not applicable on x86.
[5] Plugin not working/broken on the analyzed kernel version.
[6] Based on another plugin that uses data structures (not) identified by Fossil.
[7] Use a more complex data structure/data representation not supported by Fossil.
[8] Data structure too short to be identified/not present in the dump.

For instance, in 10 cases Fossil failed because the data structure did not contain a sufficient number of pointers to string (3 plugins) or because it was a list with less than 3 elements (7 plugins).

To our surprise, our OS-agnostic solution was in certain cases able to outperform Volatility. For instance, (highlighted cases in in Table 5.5) Fossil was able to recover the doubly-linked list of the kernel pools required by 7 Volatility plugins, which however Volatility was not able to identify. The reason, as explained in Section 5.1, is that tools that require OS-specific models (i.e., *profiles* in the Volatility jargon) may fail when the kernel is configured with an option not supported by the profile or the plugins system. At a closer inspection, our Debian image used (by default, as all latest distribution releases) the `SLUB` kernel pool system, while Volatility's plugins only supported the `SLAB` data structures.

These experiments highlight how tools that use a detailed knowledge of the internals of the OS can extract information precluded to our OS-agnostic approach. But it also shows how small divergence in the shape of atomic structs can completely blind these tools and preclude them from performing *any* analysis. This makes our OS-agnostic approach not only useful for OSs not supported by rule-based systems, but also as a complement to overcoming the limitations of existing solutions.

In a second set of experiments, we compare our approach with Virtuoso, which is an OS-agnostic forensics tool for kernel space memory proposed so far [DGSTG09]. Virtuoso collects execution traces of tailor-made test executables running on a virtualized OS and uses those traces to dynamically generate signatures to carve kernel data structures from memory dumps. The authors manually created six test executables to extract forensics-relevant information from the memory dumps of three OSs which are also present in our dataset: Linux, Windows and Haiku. By analyzing the execution traces of these test programs Virtuoso was able to extract: (i) the PID of the process running and the system time at dump time, (ii) the list of all processes including their name and PIDs, and (iii) the list of kernel modules and their names. Excepting the system time and current PID which are not represented as data structures and therefore are out-of-scope for our technique, Fossil is able to automatically extract the same information from the same OSs.

We remark that our technique has several advantages over Virtuoso. In fact, Virtuoso requires the analyst to run tailor-made executables on the target OS, run the OS in a virtualized environment and trace its entire execution multiple times. Fossil works instead on a single memory dump that can be acquired directly even from hardware devices such as phones, IoT gadgets, printers or network equipment. It also does not require any interaction with the live machine, nor special permissions to execute code or the necessity to virtualize them. Thus, we see Volatility, Virtuoso, and Fossil as complementary solutions for different tasks. Volatility is the best option for those limited targets it supports. Adding a new target is however extremely time-consuming and requires to reverse engineer the internals of the target OS. Virtuoso is the best option to *assist the development* of forensics rules for new OSs, if and only if the analyst has complete control over it and can compile and deploy new applications and trace the entire OS execution. Fossil provides instead a solution to study raw memory acquired from any class of devices and/or OSs, without the need to develop custom rules.

To summarize, rule-based systems for the limited set of use cases in which full-knowledge tools such as Volatility are available or where it is possible to interact with the OS in very specific ways, such as what Virtuoso requires, these other tools could perform better than Fossil, thanks to their "a priori" or acquired knowledge of the OS internals. For OSs in which

Table 5.6: Time required for the extraction of data structures.

OS	Minutes	OS	Minutes
Darwin	12	Linux (AArch64)	64
Embox	3	NetBSD	110
FreeBSD	26	ReactOS	11
HaikuOS	118	ToaruOS	3
HelenOS	5	vxWorks	4
iOS	22	Windows XP	12
Linux	51	Windows 10	24

rule-based tools do not work or exist, or is impossible to interact with them in a privileged way, our tool can provide useful information while the other tools are completely blind.

The median execution time to extract all data structures among the 14 OSs we analyzed in our experiments is 17 minutes (the max is 118 minutes for HaikuOS, the OS with the largest number of pointers) on an Intel Xeon 32-core CPU equipped with 128GB of RAM. Table 5.6 reports all individual execution times. Moreover, it is important to note that this operation only needs to be performed once and does not need to be repeated each time the analyst wants to explore recovered data structures. The data structure analysis then takes only a few seconds.

Finally, we do not provide an evaluation of the accuracy of the Ω function, as the function is an input to our approach and not a contribution. In particular, as discussed in 5.5, for our tests we use the function based on the output of MMUShell that not produce false negatives, i.e., it finds *all* the existing pointers in kernel memory. Our technique is then built to tolerate pointer false positives, which introduce an additional overhead but no errors in our results.

5.9 Seed-Less Data Structure Identification

We now investigate a second, more challenging scenario in which the analyst is unable to identify any seed information: this, for example, can occur if the analyst was able to dump the physical memory of the system using some hardware access that do not require the assistance of the OS running on the system (e.g. JTAG). In this case, the analysis is completely blind, but our tool can still automatically identify and reconstruct data structures from the raw dump. However, without seeds, the analyst has to manually check the extracted information to locate interesting data about the running system. To assist in this process, in a seed-less scenario, our system provides a set of OS-agnostic heuristics to filter and prioritize the number of data structures that an analyst may need to investigate.

First of all, we continue to hierarchically organize data structures as discussed in Section 5.5.4, which allows the analyst to start the exploration from structures that are less likely to be false positives. We also prioritize data structures that embed (or contain references to) printable strings, as it is simpler for human analysts to recognize information in this format and these structures can serve as entry points to explore other atomic structs linked to them.

Furthermore, for each string or reference to a string, we require that the number of unique strings is greater than 50% of the number of referenced/pointed strings: this process removes fields that contain too many repeated strings, which are unlikely to be true positives. Then, we sort the remaining data structures by the mean abundance of different strings which they embed/point, with the assumption that very frequent sequences of characters are more prone

to be false positives. Finally, when the system is able to extract global pointers from the static analysis phase, it also provides this information to the analyst.

In seed-less mode, the tool provides as output a ranked list of structures and, for each structure, it shows the address in memory of its elements and a set of strings that are either contained or referenced by pointers in the element. Table 5.7 shows examples of the forensics-relevant data structures that we are able to identify in the different OSs in a seed-less configuration. Each cell of the table reports the position in the ordered list of data structures provided by our tool. For instance, the true process list was the 2nd in the output of Darwin and the 5th for ReactOS. The "○" symbol means the tool was unable to retrieve the information in the seed-less scenario, but it succeeded when seeds were available (as shown in Table 5.3), while "-" indicates that we were not able to retrieve it at all. We have been able to identify all the process lists, the majority of kernel modules and pools lists and some filesystem-relevant data structures.

It is interesting to note that over 50% of the recovered structures appear in the top5 positions in the output of our system, and by checking the top20 candidates an analyst would be able to discover over 81% of them. This, combined with the fact that it only takes a few seconds to verify one of the output data structures, makes our OS-agnostic approach a viable solution to investigate unknown memory dumps. In fact, with the help of our tool, an analyst can quickly identify several classes of relevant information in a completely automated fashion. Of course, more time is then needed to manually inspect the memory of those structures to identify auxiliary (non-string) information, e.g., to discover which other structures are connected to the process list, which metadata are recorded for each process, to dump the memory of each process, etc.

It is important to note that in the case of seed-less analysis any comparison of Fossil with other tools results impossible because, as far as we know, it does not exist any forensics tool able to work in so extreme conditions: any information on the OS, the impossibility to instrument/virtualize it and no multiple dumps availability.

5.10 Limitations and future extensions

The data structures reconstructed by our technique are only a subset of all those that can be present in a kernel. While others could be added in the future, our system already covers all the most common data structures currently used by memory forensic tools.

Another limitation of our current implementation is that the estimation of the atomic struct size is based only on statistical properties of the surrounding memory. A possible improvement to this approach could be to use machine learning to recognize patterns among bytes located at the same offset in related atomic structs or by performing a static analysis of the kernel code in order to extract information about fields' position directly from the function creating/managing/deleting them (as done in [CMR10, PB21, FHA+22, QQY22]).

Finally, our work does not consider the presence of anti-forensics techniques that could obfuscate pointers destinations or data. An example of these, which however would have a great impact on the OS performance, is to save the destination address of critical pointers encoded with an XOR of a key located at a fixed offset in the same atomic struct. Another way is to use a not all zero words for `NULL` pointers (in a way similar to the "poisoning" technique to invalidate data). These two examples completely blind our tool and require a manual effort

Table 5.7: Ranking position of various information in the seed-less analysis results.

OS	Process list	Kernel modules	Kernel pools	Filesystem
Darwin	2	10	11	7
Embox	17	○	-	-
FreeBSD	24	31	-	26
HaikuOS	6	1	11	-
HelenOS	4	2	1	1
iOS	2	-	2	15
Linux	5	28	26	15
Linux (AArch64)	4	22	19	24
NetBSD	2	6	18	○
ReactOS	5	-	12	-
ToaruOS	3	2	3	-
vxWorks	4	-	2	-
Windows XP	5	1	2	-
Windows 10	41	○	○	○

○ Fossil is unable to retrieve the information in the seed-less scenario, but it succeeded when seeds are available.
- Fossil is not able to retrieve it in seed and seed-less scenarios.

to reverse the code which generates and manages them, invalidating the assumption of zero-knowledge about the OS.

5.11 Discussion

We discussed the problem of extracting data structures from memory dumps without any knowledge of the OS that has generated them, by extracting data structures using only their topological properties. In particular, we have posed the attention to how the presence of topological constraints may facilitate the identification of certain types of structures and, at the same time, allow to order them on the base of the reliability in their reconstruction. We have discussed how to limit the problem of false positives due to spurious chains in the Γ graphs. Furthermore, we have implemented heuristics to estimate the atomic struct sizes, a fundamental step to identify data embedded/pointed by a certain data structure.

We have tested our technique in both a seed-based and a seed-less scenarios, on 14 different OSs. In the first case, we have shown how, even with only one seed, it is possible to extract the process list, the kernel module list, the set of pools, and the information about filesystems used

by each OS. In the more challenging seed-less scenarios, an analyst was still able to extract 81% of the main data structure considered in the seed-based approach by exploring only the top 20 data structures returned by our tool.

Chapter 6

Future Works

In this thesis, we have showed how it is possible to perform a forensic analysis of an unknown operating system starting only with information about the hardware running it.

In each part of the thesis, we have always assumed that the analyst could retrieve a memory dump from the system. However, in reality, this condition often poses significant challenges. This scenario is frequently encountered in modern smartphones, which do not expose accessible hardware debug interfaces to the analyst. Additionally, these devices prevent the load of unsigned software with the necessary privileges to perform memory dumps. The problem of dump obtainment is starting to occur also on virtualization platforms as well. Technologies like AMD SEV and ARM CCA, which encrypt the memory contents of virtual machines, make the dump taken at the hypervisor level useless. This forces the analyst to access the VM directly in order to dump its volatile content. Therefore, a future line of research could investigate how often an analyst may encounter similar situations, how many operating systems effectively support and properly configure memory encryption technologies on the machine, and whether there is a possibility to exploit hardware or software vulnerabilities, such as [LZLS19, LZL21], to still perform at least a partial dump of the system's memory.

As extensively discussed, the atomicity of dumps is crucial to ensure proper forensic analysis. However, except for dumps performed by hypervisors, there are currently no tools available to perform such operations from within an operating system. Furthermore, dump tools are only available for major general-purpose operating systems like Windows, Linux, and MacOS. A potential research direction, with significant practical implications, could involve developing a universal dump software based solely on the hardware characteristics of the machine. Once loaded into memory, this software could freeze the execution of the original kernel and directly interface with the hardware to save the contents of RAM to disk or send them over the network, without any intervention from the original kernel itself. This would permit atomic memory dumps to be obtained for each operating system that supports the architecture for which the tool was developed.

We have also demonstrated how, in the presence of an MMU, it is feasible to reconstruct the mapping between virtual and physical addresses for radix tree-based architectures. However, embedded devices and real-time CPU architectures sometimes lack an MMU and rely solely on MPUs (Memory Protection Units) at best. A potential future direction of our research could therefore involve examining the possibility of identifying processes running on a system in the absence of an MMU without any knowledge about the OS. Specifically, we could investigate whether it is feasible to identify memory regions associated with different processes, also in the

case they are shared among them, distinguish among regions containing code from data regions such as the stack and heap.

In the last step of the OS-agnostic analysis that we have introduced, starting from the set of pointers, we reconstruct certain types of kernel data structures present in memory. However, for some classes of composite data structures, such as hash tables, we have not developed techniques for their automatic extraction. Although these data structures may be less common, they can still provide important information to the analyst regarding the system's state. A potential extension of our work could therefore explore how, by combining complex data structures found in the dump, such as arrays of pointers and linked lists, it is possible to help an analyst identify composite data structures, such as hash tables.

Another significant extension of our work could come from the utilization of machine learning techniques to classify available pointers and filter the extracted seeds. In the first case, assuming that the kernel utilizes memory regions of the same type (e.g., kernel pool) to allocate data structures of the same kind, we could categorize the extracted pointers to filter out all false-positive structures that are linked together by pointers belonging to different classes (and therefore pointing to objects of different types). In the second case, the filtering of seeds performed by machine learning techniques could, for example, eliminate data structures that point to or contain strings that are semantically unrelated to each other.

The data structure recognition technique that we have developed permits to recognize kernel data structures. However, since the same types of data structures are often used within libraries and entire user-space applications, this technique could be applied to forensic analysis of memory dumps from user-space applications as well. This would, not only permit the memory of unknown applications to be analyzed without having to reverse engineer much of the code, but also permit to implement automatic carving techniques based on topology constraints to identify deallocated and no longer referenced data structures within process memory.

In conclusion, this thesis has shown that is possible to perform, at least a basic, memory forensics analysis of an operating systems without known anything about its internal, but leaverage only informaion about the underlying hardware, its constraints and universal topological properties of data structures. The zero-knowledge approach is not intended to replace existing forensics techniques for well-known, well-studied operating systems but to work alongside them to help analysts in all those cases where more specific tools are not available. We hope that in the future, the techniques developed in this thesis will be integrated into currently existing open-source tools and that our work will push the memory forensics community to investigate further the topological properties of data structures, the relationships between their organization and the underlying hardware architecture, and the use of techniques not based on a deep knowledge of individual operating systems but rather on the characteristics common to all of them.

Appendices

Résumé en français

La Forensique Numérique

Nous vivons à une époque de technologie de l'information, où les dispositifs numériques sont essentiels à nos activités quotidiennes. Les ordinateurs, les téléphones mobiles, les réseaux et les dispositifs d'infrastructure, les systèmes embarqués et même les appareils domestiques connectés sont devenus omniprésents dans la société moderne, jouant un rôle vital. Nous comptons sur ces dispositifs pour stocker, traiter et transmettre nos données personnelles et professionnelles, effectuer des paiements, et nous leur confions même notre sécurité, par exemple lors d'un vol en avion ou de l'utilisation d'une voiture autonome. Cependant, ces dispositifs peuvent être la cible d'acteurs malveillants qui cherchent à les compromettre à leur propre avantage. Ces acteurs peuvent accéder illégalement à ces dispositifs et manipuler leur contenu ou leur fonctionnalité, ce qui peut avoir de graves conséquences pour les individus, les organisations et la société en général, entraînant des pertes financières, des atteintes à la sécurité nationale ou même mettant des vies en danger.

Étant donné qu'aucun système ne peut être totalement à l'abri des cyberattaques, il ne suffit pas de compter uniquement sur des défenses préventives. Il est également nécessaire de détecter et de répondre à de tels incidents de manière rapide et efficace en reconstituant la séquence d'événements que l'attaquant a réalisé. Cela implique d'identifier les vulnérabilités exploitées, les actions effectuées et l'impact causé par l'attaque, ainsi que de récupérer les artefacts utilisés par l'attaquant. Tout cela fait partie de la discipline de la forensique numérique, qui consiste à appliquer des méthodes et des techniques scientifiques pour collecter et examiner des preuves numériques de manière scientifiquement valable. L'un des premiers exemples de forensique numérique était axé sur l'analyse des journaux et de l'état des systèmes en direct. C'était le cas en 1989, lorsque Clifford Stoll, astronome et administrateur système au Lawrence Berkeley National Laboratory, a découvert un espion ouest-allemand qui s'était infiltré dans son réseau et volait des données pour les vendre à l'Union soviétique [Sto89].

La forensique numérique a connu des avancées significatives dans les années 90 et 2000, englobant non seulement l'analyse des journaux système, mais aussi l' analyse des disques et des réseaux. L'analyse des disques consiste à extraire des données brutes à partir de dispositifs de stockage non volatils, tels que les disques durs, afin d'identifier les fichiers actifs, modifiés ou supprimés. Elle consiste également en la recherche des données cachées ou chiffrées qui peuvent fournir des informations précieuses sur les activités, les préférences, les communications et les intentions de l'utilisateur. L'analyse forensique du réseau se concentre quant à elle sur la capture et l'analyse des paquets de données réseau pour reconstituer les événements, détecter les intrusions potentielles et découvrir des preuves d'activités malveillantes. En examinant les schémas de trafic réseau, les spécialistes de l'analyse forensique du réseau obtiennent des

informations cruciales sur les actions entreprises à la fois par les attaquants et les victimes, facilitant ainsi la reconstitution des événements et une compréhension approfondie de l'ampleur de la violation de sécurité.

Ces dernières années, le domaine de la forensique numérique s'est étendu pour inclure les domaines de la téléphonie mobile et de l'informatique en cloud. Les appareils mobiles, tels que les smartphones, les tablettes et les appareils portables, stockent désormais une quantité considérable de données personnelles et sensibles, qui revêtent une importance pour diverses finalités. Dans ce paysage en constante évolution, l'analyse forensique de la mémoire émerge comme un élément essentiel des enquêtes numériques.

L'Analyse Forensique de la Mémoire

L'analyse forensique de la mémoire, en particulier, est la branche de la forensique numérique qui traite de l'analyse de la mémoire volatile d'un système informatique. Elle est apparue en tant que domaine de recherche en réponse à la sophistication croissante et à la furtivité des logiciels malveillants et des cyberattaques, lorsque les techniques de l'analyse forensique des disques n'étaient pas suffisantes pour détecter et analyser les logiciels malveillants qui ne laissaient pas de traces sur les dispositifs de stockage non volatils.

La mémoire volatile, également connue sous le nom de mémoire à accès aléatoire (RAM), est la mémoire principale d'un système et est utilisée par le système d'exploitation (OS) et les applications de l'espace utilisateur pour stocker temporairement et traiter des données. En collaboration avec les applications de l'espace utilisateur et le code du noyau, la RAM contient des structures de données du noyau qui conservent des informations sur l'état du système et peuvent être analysées pour extraire des artefacts d'exécution tels que la liste des processus en cours d'exécution, les modules du noyau chargés, les connexions réseau, les clés de chiffrement, les signatures de logiciels malveillants, et bien plus encore. Ces artefacts peuvent fournir des informations précieuses pour les enquêtes criminelles et les interventions en cas d'incident, car ils peuvent révéler l'état et l'activité du système au moment de l'acquisition de la mémoire.

Une analyse forensique de la mémoire se compose de trois phases principales : l'acquisition, l'interprétation et l'analyse.

- **Acquisition** Dans cette phase, le contenu de la mémoire est acquis à partir du système cible à l'aide d'outils et de techniques spécialisés. Le processus d'acquisition doit être effectué de manière scientifiquement valable, ce qui signifie qu'il ne doit pas altérer ou endommager les données originales de la mémoire, et il doit préserver l'intégrité et l'authenticité des éléments de preuve.

- **Interprétation** Cette phase consiste à identifier et extraire les artefacts pertinents à partir du fichier image de la mémoire à l'aide d'un outil ou d'un cadre d'investigation numérique. Le processus d'interprétation nécessite une connaissance approfondie des structures de données et des formats utilisés par le système d'exploitation et les applications pour stocker des informations en mémoire.

- **Analyse** L'analyste examine et corrèle les artefacts pour reconstituer les événements et les activités qui se sont produits sur le système cible. Cette phase nécessite une combinaison de compétences techniques et de connaissances spécifiques pour interpréter et évaluer les éléments de preuve.

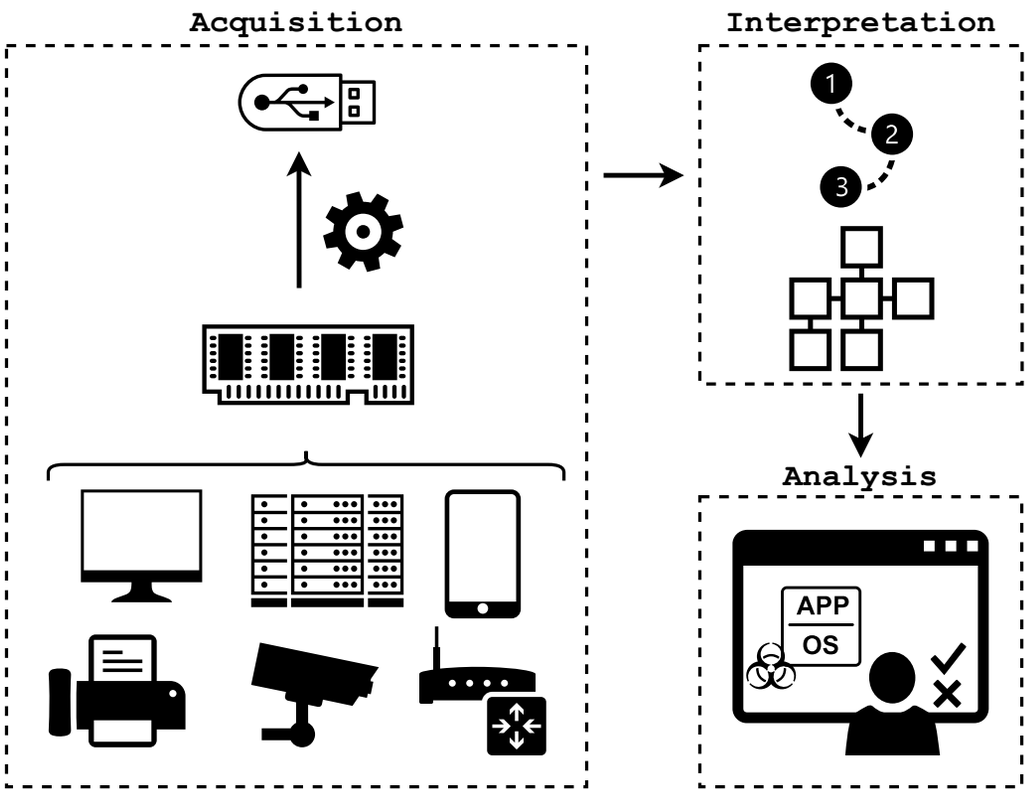


Figure 1: Les trois principales phases d'une analyse forensique de la mémoire sont : l'acquisition de la mémoire principale, l'interprétation des données dans le dump de la mémoire et l'analyse des artefacts.

Deux caractéristiques de l'analyse forensique de la mémoire en particulier posent de nombreux défis aux chercheurs en sécurité et aux analystes en criminalistique : la mémoire volatile est transitoire et son contenu est continuellement modifié par le système d'exploitation et les applications utilisateur, et contrairement à un dump d'un disque dur où les données sont organisées selon une structure de système de fichiers bien reconnaissable, les données dans un dump de la mémoire ne présentent pas de frontières claires et de sémantique.

Problèmes Ouverts en Analyse Forensique de la Mémoire

Considérez les informations sur les processus en cours d'exécution sur un système : lorsque le noyau lance un nouveau processus, il doit conserver le nom du processus, sa disposition en mémoire, les bibliothèques chargées et autres métadonnées pour suivre correctement le processus pendant son exécution et être en mesure de libérer les ressources allouées lorsqu'il est terminé. Un analyste qui souhaite récupérer ces informations à partir d'un dump mémoire doit, en premier lieu, collecter le contenu de la mémoire en le sauvegardant sur un support non volatile (phase d'acquisition).

Cette tâche, qui semble en surface anodine, s'avère être remplie de pièges et de problèmes cachés. Tout d'abord, différents types de machines et d'architectures de processeurs nécessitent

différentes techniques pour effectuer le dump, et chacune d'entre elles est porteuse de problèmes supplémentaires. Considérons, par exemple, le cas où l'analyste a un accès direct au matériel de la machine : dans ce scénario, on pourrait utiliser des interfaces de débogage sur la carte mère (comme l'interface JTAG [RALJA21]) qui permettent d'arrêter le processeur et de lire directement le contenu de la mémoire physique. Cependant, ces interfaces sont souvent désactivées par les fabricants, comme c'est le cas pour la plupart des smartphones, ou elles ne sont même pas accessibles sauf par le biais d'exploits ou nécessitent des niveaux d'autorisation détenus uniquement par le fabricant du processeur (comme c'est le cas des interfaces DCI sur les plateformes INTEL [LSF21]). Cependant, il existe d'autres voies d'accès matériel qui pourraient être exploitées pour obtenir un dump, telles que l'utilisation des contrôleurs de gestion de la carte mère (BMCs) (lorsqu'ils sont présents) [LBF20], ou l'utilisation de fonctionnalités intégrées dans le micrologiciel du système, comme les Firmware Assisted Dumps dans le cas des plateformes IBM POWER [pow23]. Si le système a été conçu avec l'idée de pouvoir effectuer un dump de la mémoire en cas de besoin, il est possible qu'une interface PCI-E permettant un accès externe et direct à la mémoire RAM ait été installée, ou que l'autorisation d'effectuer un transfert DMA via USB4/Thunderbolt [LPF19] ait été activée au démarrage. Dans presque tous les cas, cependant, il n'est pas possible de réaliser un dump mémoire via le matériel, et l'analyste doit utiliser un logiciel disposant d'un niveau de privilège de sécurité qui lui permet de lire toute la mémoire physique du système. Ce logiciel de capture peut être chargé par l'analyste au moment du dump, tel qu'un module noyau, ou être déjà présent dans le système, comme c'est le cas de SMM [RFP+12] ou des modules UEFI [LHKF21]. Cependant, la méthode d'acquisition logicielle présente deux problèmes très subtils qui peuvent compromettre irrémédiablement le dump. Le premier concerne l'utilisation d'un composant logiciel résidant en mémoire RAM et utilisant cette mémoire pour son fonctionnement, ce qui altère le contenu de la mémoire elle-même (problème équivalent à l'effet observateur en mécanique quantique). Le deuxième problème concerne l'atomicité du dump : le processus de dump n'est pas instantané et une certaine quantité de temps est nécessaire pour copier chaque page physique de la mémoire vers un support non volatile ou l'envoyer via le réseau. La plupart des outils de dump (à l'exception de l'utilisation des capacités des hyperviseurs dans le cas du dump mémoire de machines virtuelles qui n'utilisent pas les technologies de chiffrement VM comme AMD SEV) n'arrêtent pas le noyau et les processus utilisateur pendant le dump, les laissant libres de continuer à écrire et à modifier la mémoire. Les dumps obtenus de cette manière présentent des régions de mémoire dumpées à des moments différents qui contiennent des pointeurs vers des données qui ont depuis été modifiées ou qui ne sont plus présentes au moment du dump, voire des structures de données différentes occupant de manière aléatoire la même portion de mémoire.

Après l'acquisition, l'analyste doit traiter un ensemble de données qui représente le contenu de la mémoire physique du système, car contrairement à un dump d'un disque dur où les données sont organisées selon une structure de système de fichiers bien reconnaissable, les données dans un dump mémoire ne possèdent pas de frontières claires et de sémantique. Pour corréler les artefacts contenus dans le dump de la mémoire physique (phase d'analyse), l'analyste doit d'abord les révéler en comblant deux écarts sémantiques distincts (phase d'interprétation) : la traduction des adresses virtuelles en adresses physiques, qui consiste à traduire les pointeurs exprimés sous forme d'adresses virtuelles vers leur position physique dans la mémoire, et la détection des structures de données du noyau, par exemple celles liées aux processus en cours d'exécution, à la gestion de la mémoire et aux modules du noyau. Pour résoudre ces deux

problèmes, un analyste doit connaître les détails internes du noyau du système d'exploitation. Pour les systèmes d'exploitation bien connus tels que Windows, Linux et macOS, des outils classiques de forensics tels que Volatility [Wal17] et Rekall [Coh14] disposent, pour chaque système d'exploitation pris en charge, de collections de *profils*. Un *profil* est une sorte de "carte" qui décrit, pour une version et une configuration spécifiques d'un système d'exploitation donné, l'emplacement des structures de données du noyau en mémoire, leur taille ainsi que le type et l'emplacement de chaque champ les composant. Grâce aux *profils*, en utilisant des heuristiques et des règles spécifiques à chaque système d'exploitation, ces outils peuvent récupérer la correspondance virtuel-physique et extraire des informations forensiques pertinentes à partir du dump mémoire analysé. Cependant, la collection de *profils* doit être constamment mise à jour, car les structures de données utilisées par un système d'exploitation peuvent évoluer avec le temps. De plus, pour des systèmes d'exploitation tels que Linux, qui permettent un haut degré de personnalisation de la configuration du noyau, il peut même être nécessaire de créer un profil spécifique pour chaque machine que l'analyste souhaite investiguer. Certains travaux récents [PB21, QQY22, FHA+22] ont partiellement surmonté le problème de la nécessité de *profils* spécifiques pour les configurations personnalisées du noyau Linux, en les reconstituant directement à partir des dumps mémoire ou en effectuant une analyse forensique sans leur utilisation. Cependant, même dans ces travaux, les auteurs supposent avoir une connaissance approfondie des détails internes du noyau et être en mesure d'identifier des fonctions spécifiques capables de fournir des indices sur les structures de données.

L'approche classique de l'analyse forensique de la mémoire est étroitement liée à la connaissance des détails internes du système d'exploitation, ce qui pose un défi sérieux pour l'avenir de l'analyse forensique de la mémoire. En effet, l'augmentation rapide des appareils embarqués, des objets IoT et des machines virtuelles hébergées sur le cloud se traduit par un nombre croissant de systèmes d'exploitation et d'architectures de processeurs variées, qui en général ne sont pas pris en charge par les outils forensiques actuels et nécessitent un effort considérable pour être étendus. De nos jours, un analyste qui se retrouve avec un dump mémoire d'un système d'exploitation peu courant, voire inconnu, ne peut compter que sur des outils basiques pour extraire, par exemple, des chaînes brutes, et doit effectuer un travail approfondi de rétro-ingénierie du binaire du noyau (lorsqu'il est disponible) pour être capable d'extraire davantage de données structurelles. Pour toutes les raisons mentionnées ci-dessus, l'analyse forensique de la mémoire sur des systèmes d'exploitation peu courants, pour lesquels la traduction des adresses virtuelles en adresses physiques et/ou l'organisation des structures de données du noyau ne sont pas connues ou pour lesquels aucun *profil* n'est disponible, devient impossible, mettant en évidence toutes les limites de cette approche et empêchant ainsi son applicabilité sur plusieurs systèmes d'exploitation.

Sujets de Recherche

Pour surmonter ce problème, nous introduisons, pour la première fois, le concept d'*analyse forensique de la mémoire à connaissance nulle* : réaliser une analyse forensique de la mémoire sans *aucune* connaissance du système d'exploitation sous-jacent. Bien que les profils, les règles personnalisées et l'inspection dynamique restent des solutions précieuses qui sont susceptibles de fournir de meilleurs résultats lorsqu'ils peuvent être appliqués, nous estimons que notre nouvelle approche est nécessaire pour étendre rapidement l'analyse de la mémoire à une classe plus

large de systèmes cibles. En supposant que nous ayons effectué un dump mémoire du système d'exploitation inconnu, en utilisant uniquement les informations dérivées de la configuration matérielle de la machine, nous affirmons qu'il est possible de reconstruire l'espace d'adressage du noyau de manière indépendante du système d'exploitation, puis d'identifier les pointeurs du noyau. À partir de ceux-ci, nous affirmons également qu'il est possible de reconstruire les structures de données du noyau en mémoire telles que les listes et les arbres en utilisant uniquement des propriétés indépendantes du système d'exploitation liées à leur topologie. Cependant, pour prouver la faisabilité de notre théorie, nous devons répondre à trois questions:

Question #1: Est-il possible de *quantifier* dans quelle mesure le processus de dump et la non-atomicité des dumps mémoire affectent la récupérabilité et la cohérence des espaces d'adresses virtuelles et des structures de données du noyau ?

Comme indiqué dans les sections précédentes, la plupart des outils d'imagerie mémoire basés sur des logiciels présentent un problème : ils n'arrêtent pas le système pendant l'acquisition de la mémoire. Cela entraîne une acquisition non atomique qui comporte des incohérences, car le dump n'est pas une représentation unitaire du contenu de la mémoire à un moment spécifique, mais ressemble plutôt à une longue exposition photographique de celle-ci.

Dans le cas d'un système d'exploitation inconnu, pour lequel nous ne savons rien, y compris la façon dont il utilise la mémoire physique, la manière la plus simple et naturelle de le dumper est de dumper les pages physiques séquentiellement selon leur adresse physique. Incidemment, c'est également la stratégie utilisée par les outils de dump open-source disponibles pour les principaux systèmes d'exploitation (comme LiME [Ben23] pour Linux ou Winpmem [win23] pour Windows).

Plusieurs articles de recherche ont décrit le problème de non-atomicité [VF12, GF16, PFB19], mais l'impact de l'outil de dump lui-même sur la RAM et les effets de la non-atomicité sur la disponibilité, l'accessibilité et la cohérence des espaces d'adresses virtuelles et des structures de données du noyau n'ont jamais été quantifiés.

Question #2: Est-il possible de reconstruire la correspondance des adresses virtuelles vers physiques sans *aucune* connaissance préalable sur le système d'exploitation, en se basant uniquement sur le dump mémoire et les informations sur le matériel ?

La première étape de la phase d'identification consiste en la traduction des adresses virtuelles en adresses physiques. Si le système d'exploitation est connu, il existe, comme mentionné précédemment, des heuristiques qui permettent de déduire la correspondance entre les adresses. Cependant, si le système d'exploitation est inconnu, ces heuristiques ne sont pas disponibles et les outils forensiques couramment utilisés tels que Volatility [Wal17] sont inutiles. De plus, le problème de la reconstruction des espaces d'adresses dans la littérature est présenté comme étant "résolu a priori", sans jamais entrer dans les détails du problème. Dans le cas d'un système d'exploitation inconnu, il faut se fier uniquement au matériel de la machine (qui est supposé être connu et documenté) car ses caractéristiques ne dépendent pas du système d'exploitation. Dans ce cas également, la littérature est lacunaire, car les quelques cas qui abordent le problème de la reconstruction de l'espace d'adresses en utilisant des informations matérielles, comme [SG10], se limitent toujours à l'architecture x86, ignorant les architectures de processeurs avec

des mécanismes de traduction d'adresses virtuelles différents, même dans le cas de systèmes d'exploitation connus.

Question #3: Est-il possible d'extraire les structures de données du noyau telles que les listes chaînées, les tableaux, etc., sans *aucune* connaissance du système d'exploitation, en se basant uniquement sur le dump mémoire et les informations sur le matériel ?

La deuxième et dernière étape de la phase d'identification consiste à reconnaître les structures de données du noyau qui contiennent des données pertinentes pour l'analyse forensique. Pour un système d'exploitation bien connu, un analyste peut utiliser un *profil* pour guider les outils forensiques dans la localisation et l'exploration des structures de données du noyau. Il s'avère donc que dans le cas d'un système d'exploitation inconnu, l'approche par *profil* n'est pas applicable et l'analyste est complètement dans l'obscurité. Dans la littérature, certains travaux ont tenté de pallier le besoin de *profils* en utilisant, par exemple, des machines virtuelles instrumentées exécutant le système d'exploitation en question [DGSTG09, LZX10] pour déduire des informations sur les structures de données utilisées par le noyau, ou en analysant son code source [CMR10, FPW+16, LRW+12, LRZ+11]. Toutes ces techniques supposent cependant de disposer de données *"auxiliaires"* sur le système d'exploitation ou de capacités spéciales, telles que l'accès au code source ou la possibilité de l'exécuter/héberger sur une machine virtuelle. Nous souhaitons, en revanche, déduire les structures de données du noyau pertinentes pour l'analyse forensique uniquement à partir du dump mémoire de la machine, sans aucune autre information sur le système d'exploitation.

Contributions

"Is The End of Memory Acquisition As We Know It?"

Dans ce travail, nous présentons un système basé sur un système d'émulation record-replay PANDA [DGHH+15] pour suivre toutes les opérations d'écriture effectuées par le noyau du système d'exploitation au cours d'un processus d'acquisition de mémoire. Cela nous permet de quantifier, pour la première fois, le nombre et le type d'incohérences observées dans les dumps de memoire. Nous examinons l'activité de trois systèmes d'exploitation différents au repos et la façon dont ils gèrent la mémoire physique. Nous examinons l'activité de trois systèmes d'exploitation différents en idle et la manière dont ils gèrent la mémoire physique. Ensuite, en nous concentrant sur Linux, nous quantifions comment différents modes de dump, systèmes de fichiers et cibles matérielles influencent la fréquence des écritures du noyau pendant le dump. Nous analysons également l'impact des incohérences sur la reconstruction des tables de pages et des structures de données noyau utilisées par Volatility pour extraire des forensics artifacts. Nos résultats montrent que les incohérences sont courantes et peuvent compromettre la fiabilité et la validité de l'analyse de dumps de mémoire effectués de manière non atomique. incohérences peuvent compromettre les résultats présentés à un analyste.

Ce projet a bénéficié du soutien du Conseil européen de la recherche (ERC) dans le cadre du programme de recherche et d'innovation Horizon 2020 de l'Union européenne, accord de subvention n° 771844 (BitCrumbs). L'outil développé dans le cadre de ce travail est publié en

tant que projet open-source, ainsi que le jeu de données non soumis à des restrictions de licence spécifiques au système d'exploitation.

Cette contribution est actuellement soumise à l'examen de la revue ACM Transactions on Privacy and Security (TOPS).

"In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics"

Dans ce travail, nous explorons pour la première fois le concept de combler l'écart sémantique dans les traductions virtuelles vers physiques sur 10 architectures de processeur différentes, de manière indépendante du système d'exploitation. Nous étudions les contraintes fondamentales imposées par l'unité de gestion de mémoire (MMU) qui peuvent être utilisées pour construire des signatures permettant de récupérer les structures de données requises à partir de la mémoire, sans nécessiter de connaissances préalables sur le système d'exploitation sous-jacent. De plus, nous introduisons une étape d'analyse de code statique pour récupérer la configuration des registres MMU définie par le système d'exploitation au démarrage. À titre de preuve de concept, nous développons un outil appelé MMUShell, qui s'appuie uniquement sur des paramètres dérivés de l'architecture d'ensemble d'instructions du processeur (ISA) pour reconstruire les correspondances virtuelles-physiques. Nous menons des expériences pour extraire des espaces d'adresses virtuelles, mettant en évidence les défis liés à la traduction d'adresses virtuelles en adresses physiques de manière agnostique vis-à-vis du système d'exploitation dans des scénarios réels. Nous évaluons notre approche sur un ensemble diversifié de 26 systèmes d'exploitation différents et présentons une utilisation pratique sur un périphérique matériel réel. De plus, nous démontrons comment notre technique peut être utilisée pour récupérer et analyser les processus de l'espace utilisateur s'exécutant sur un système d'exploitation inconnu, même sans aucune connaissance interne du système. Cela permet aux analystes forensiques d'initier des enquêtes et de réaliser des analyses sur de tels systèmes.

Ce projet a bénéficié du soutien du Conseil européen de la recherche (ERC) dans le cadre du programme de recherche et d'innovation Horizon 2020 de l'Union européenne, accord de subvention n° 771844 (BitCrumbs) et accord de subvention n° 786669 (ReAct). L'outil développé dans le cadre de ce travail est publié en tant que projet open-source, ainsi que le jeu de données non soumis à des restrictions de licence spécifiques au système d'exploitation.

Cette contribution a été publiée dans un article paru dans la revue ACM Transactions on Privacy and Security (TOPS) [OB22]

"An OS-Agnostic Approach to Memory Forensics"

Dans ce travail, nous introduisons le nouveau concept de l'analyse forensique de la mémoire à connaissance nulle, basée sur des techniques qui permettent de récupérer certaines informations forensiques sans aucune connaissance interne du système d'exploitation sous-jacent. Notre approche permet d'identifier automatiquement différents types de structures de données en utilisant uniquement leurs contraintes topologiques, puis prend en charge deux modes d'investigation. Dans le premier mode, il permet de parcourir les structures récupérées à partir de *semences* prédéterminées, c'est-à-dire des informations pertinentes pour la forensique (telles qu'un nom de processus ou une adresse IP) que l'analyste connaît a priori ou qui peuvent être facilement identifiées dans le dump. Nous avons implémenté notre technique dans un outil de preuve de concept appelé Fossil, qui nous a permis de mener diverses expériences dans des

scénarios réels. Nos expériences montrent qu'une seule semence peut suffire à récupérer la liste complète des processus et d'autres structures de données forensiques importantes dans des dumps obtenus à partir de 14 systèmes d'exploitation différents, sans aucune connaissance des noyaux sous-jacents. Dans le deuxième mode de fonctionnement, notre système ne nécessite aucune semence, mais utilise plutôt un ensemble d'heuristiques pour classer toutes les structures de données en mémoire et présenter aux analystes seulement les plus prometteuses. Même dans ce cas, nos expériences montrent qu'un analyste peut utiliser notre approche pour identifier facilement des informations structurées pertinentes pour la forensique dans un scénario véritablement agnostique vis-à-vis du système d'exploitation.

Ce projet a bénéficié du soutien du Conseil européen de la recherche (ERC) dans le cadre du programme de recherche et d'innovation Horizon 2020 de l'Union européenne, accord de subvention n° 771844 (BitCrumbs). L'outil développé dans le cadre de ce travail est publié en tant que projet open-source, ainsi que le jeu de données non soumis à des restrictions de licence spécifiques au système d'exploitation.

Cette contribution a été présentée lors du 30ème Network and Distributed System Security Symposium (NDSS) [ODB23]."

Recherches Afférents

Quantification des incohérences dans les extraits de mémoire non atomiques

Au fil du temps, de nombreuses études [Kor07, LK08, HBN09, MC13] ont souligné comment le manque d'atomicité dans les extraits de mémoire peut entraîner des problèmes et des erreurs lors d'une analyse forensique. Les premiers à formaliser le concept d'extraits atomiques étaient, en 2010, *Vomel et Freiling* [VF12] qui ont également introduit la distinction entre l'intégrité, la correction et l'atomicité d'un extrait. Dans un travail ultérieur, *Gruhn et Freiling* [GF16] ont évalué 12 outils d'acquisition différents qui respectaient ces trois critères. En 2021, *Freiling et al.* ont étendu le concept d'atomicité et d'intégrité également aux instantanés de disque dans [FG21]. Plus récemment, Case et Richard ont souligné le problème pressant de la diffusion des pages [CRI17]. Avec la prévalence des serveurs équipés de grandes quantités de RAM entraînant des temps d'acquisition plus longs, l'effet de diffusion des pages est devenu de plus en plus courant. En 2019, *Pagani et al.* [PFB19] ont introduit la "dimension temporelle" dans l'analyse forensique de la mémoire. Cette dimension offre à l'analyste un moyen initial d'évaluer l'atomicité des structures de données utilisées lors d'une analyse. De plus, une série d'expériences a été réalisée pour démontrer que la diffusion des pages introduit non seulement des incohérences dans les tables de pages, mais affecte également de manière indiscriminée toute analyse impliquant des structures situées dans l'espace utilisateur et noyau.

Parmi les premiers à proposer une solution au problème de non-atomicité des extraits de mémoire, avant même la formalisation de la définition, *Huebner et al.* ont proposé dans leur travail [HBHW07] l'acquisition automatique de l'état du noyau et des applications utilisateur effectuée périodiquement par le système d'exploitation. Cette solution nécessite cependant une refonte des mécanismes internes du noyau, ce qui n'est pas immédiatement applicable aux systèmes d'exploitation existants.

Une approche intéressante pour obtenir un instantané atomique par acquisition logicielle a

été développée par *Schatza et Bradley* [Sch07]. À l'exécution, ils injectent un noyau minimal qui arrête l'exécution du système d'exploitation en cours et extrait sa mémoire. Cette approche, comme souligné par les auteurs, présente plusieurs limitations en raison de l'intégration étroite entre le noyau d'extraction et le système d'exploitation d'origine, ainsi que de la variabilité du matériel qui doit être pris en charge.

En 2009, Forenscope [HSH+09] a utilisé une approche basée sur l'effet de rémanence des données dans les puces mémoire après le redémarrage du système pour acquérir la mémoire de manière atomique. Cette technique, connue sous le nom d'acquisition de mémoire à froid, a malheureusement montré une forte dépendance à l'égard du jeu de puces et des modules mémoire utilisés, certaines combinaisons de composants ne parvenant pas à conserver le contenu de la RAM après le redémarrage [CBS11].

L'année suivante, *Martignoni et al.* ont introduit HyperSleuth [MFPC10], qui utilise un hyperviseur personnalisé injecté à l'exécution pour effectuer des extraits de mémoire en utilisant la technique du dump-sur-écriture et du dump-en-attente afin de résister aux logiciels malveillants évasifs qui peuvent altérer le noyau du système d'exploitation, tout en obtenant un extrait atomique. Cependant, cette technologie prometteuse nécessite l'activation du support de virtualisation sur le processeur avant le démarrage de la machine, et n'est pas applicable dans les cas où le système d'exploitation lui-même agit en tant qu'hyperviseur, comme dans le mode virtuel sécurisé de Windows 10.

Récemment, dans l'article *"Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots"* [FHA+22], les auteurs ont introduit parallèlement un module noyau Linux permettant d'obtenir des extraits de mémoire atomiques. Cependant, la technique développée, en bloquant temporairement les interruptions système, peut provoquer de multiples plantages en chaîne des modules noyau responsables de la gestion matérielle à la fin du processus d'extraction.

D'autres solutions développées pour obtenir des extraits atomiques impliquent l'utilisation de micrologiciels modifiés pour exécuter l'outil d'extraction avec des niveaux de privilège plus élevés (comme le module SMM [RFP+12]) ou avec le système d'exploitation arrêté mais avec le contenu de la mémoire encore disponible (comme un module EFI [LHKF21]). Cependant, ces deux méthodes nécessitent une modification en profondeur du micrologiciel système résidant sur la carte mère ou son installation dans la partition EFI, ainsi qu'une signature cryptographique valide pour permettre à l'outil d'extraction de s'exécuter dans un environnement Secure Boot, qui est désormais une condition par défaut pour les systèmes d'exploitation modernes sur les processeurs Intel.

Reconstruction des espaces d'adresses virtuelles et identification des structures de données

La récupération de structures de données agnostiques au système d'exploitation, en tant qu'application particulière du problème plus général de la reconstruction de données, a été explorée par divers auteurs en utilisant différentes techniques. Certaines de ces techniques sont dérivées de la forensique traditionnelle des disques. Un exemple classique est la récupération de données, dans laquelle l'analyste tente de récupérer des informations à partir du dump mémoire en utilisant des signatures qui identifient de manière unique les données d'intérêt. Par exemple, *Beverly et al.*[BGC11] créent un ensemble de signatures pour récupérer des paquets réseau à partir d'un dump mémoire, tandis que *Van Barr*[vAv08] a développé une technique pour récupérer

des fichiers mappés en mémoire à partir de dumps Windows. Une approche plus générique est proposée par *Wagner et al.* [WRG15], qui reconstruisent les structures de données de bases de données relationnelles dans des dumps mémoire pour différents moteurs SQL.

Saur and Grizzard[SG10] sont les premiers à proposer une approche de reconstruction de radix trees pour Intel x86-32 à partir de dumps mémoire de Windows XP et Linux. Leur technique d'analyse est basée sur des règles dérivées des spécifications de l'ISA et des heuristiques sur mesure dérivées des internes des systèmes d'exploitation, ce qui permet aux auteurs de récupérer les espaces d'adresses virtuelles et physiques des processus cachés s'exécutant sur un système échantillonné. Leur approche est cependant limitée par la non-universalité de leurs règles d'analyse, qui contiennent des heuristiques basées sur la façon dont les noyaux Linux et Windows utilisent les tables de pages, ce qui rend impossible leur utilisation sur des dumps de différents systèmes d'exploitation. Un travail le plus étroitement lié à notre étude est l'article parallèle de *Tran-Quoc et al.*[TQHMT21]. L'article se concentre principalement sur la reconstruction des tables de pages pour Intel x86 uniquement, mais les auteurs décrivent également un cas d'utilisation simple dans lequel ils utilisent les pointeurs récupérés pour identifier la liste des processus en cours d'exécution dans Linux, BSD et MS Windows. Bien que l'idée d'identifier et de suivre les pointeurs soit similaire à notre technique, ce travail ne traite pas de la reconstruction de structures de données, ne peut pas gérer les arbres, les tableaux ou les cas où le nom du processus ne fait pas partie de la structure de liste chaînée elle-même, sans considérer également l'analyse sans semence. Un travail plus similaire au nôtre pour sa focalisation sur l'ISA du CPU mais avec un objectif différent est [GLB13]. Dans ce travail, les auteurs utilisent des règles d'analyse agnostiques au système d'exploitation pour détecter, dans les dumps mémoire, les structures de contrôle des machines virtuelles utilisées par les processeurs Intel pour maintenir l'état des machines virtuelles.

Le problème plus général de l'extraction de structures de données à partir de dumps mémoire a déjà été abordé dans plusieurs études précédentes, même si avec des objectifs différents. Par exemple, *Dolan-Gavitt et al.* [DGSTG09] ont proposé une approche basée sur des signatures pour définir les invariants des structures de données du noyau et générer des signatures pour leur récupération à partir des dumps mémoire. Une approche similaire, mais basée également sur des inférences probabilistes, a été développée dans [LRW+12]. *Lin et al.* ont également développé REWARDS [LZX10], qui instrumente uniquement les processus de l'espace utilisateur en utilisant Intel Pin, capture les horodatages de chaque accès mémoire et révèle leurs structures de données dans les dumps mémoire du système. Un autre travail basé sur des signatures est "Multi-Aspect, Robust, and Memory Exclusive Guest OS Fingerprinting" [GFP+14]. Dans ce travail, *Yufei et al.* utilisent des invariants, dérivés du code du noyau et des structures de données extraites d'un dump mémoire d'un système exécutant Linux, pour générer des signatures permettant d'identifier la version du noyau. SigGraph [LRZ+11], quant à lui, reconstruit le graphe des structures de données à l'intérieur d'un dump mémoire d'un système d'exploitation générique en utilisant des signatures dérivées de son code source. Une approche basée sur plusieurs instantanés de la mémoire du même processus pour collecter des informations sur les structures de données utilisées par l'application est explorée par *Urbina et al.* [UGCL14] et par *Feng et al.* [FPYL14]. (DeepMem [SYLS18]), quant à lui, génère des représentations abstraites pour les objets du noyau en utilisant une approche d'apprentissage profond basée sur les graphes, ce qui nécessite cependant plusieurs dumps du même système d'exploitation pour être entraîné, compromettant son extensibilité aux systèmes d'exploitation inconnus. D'autres travaux se sont plutôt concentrés sur la reconstruction de la forme, du contenu et des types

primitifs des structures de données [MCJ17, SSB10, SSB11, TDC10].

Les chercheurs ont également étudié comment créer des signatures à partir de fichiers binaires exécutables : ORIGEN [FPW+16] reconstruit les décalages des structures de données atomiques en utilisant une analyse statique sur le code d'une version précédente du même noyau, tandis que *Case et al.* [CMR10] reconstruisent les décalages des structures Linux importantes en analysant directement le code intégré dans le dump lui-même. Une approche différente basée sur des inférences logiques sur la position des champs de la structure permet à *Qi et al.* [QQY22] de dériver le décalage des champs importants dans les structures de données du noyau Linux. Cependant, si les emplacements des champs sont aléatoires lors de la compilation, ils ne peuvent pas les récupérer. Deux articles récents et indépendants [PB21, FHA+22] ont développé une technique basée sur une combinaison d'analyse symbolique et d'émulation de code, qui permet de récupérer les décalages des structures même en cas de leur randomisation. Ils peuvent extraire un profil de Volatility à partir d'un dump mémoire du noyau Linux sans aucune autre information, permettant l'analyse même dans le cas où il est impossible de le générer en temps d'exécution.

Toutes ces techniques travaillent sur des structures de données contenant des adresses virtuelles, supposant ainsi que le problème de la traduction des espaces d'adresses a déjà été résolu par d'autres moyens, ou elles nécessitent plusieurs dumps de la même version du système d'exploitation pour entraîner un algorithme. De plus, elles supposent que l'analyste connaît le système d'exploitation et quelque chose sur son fonctionnement interne (accès au code source, définitions des structures de données à récupérer, plusieurs dumps du même système dans différentes conditions, etc.), utilisent des approximations adaptées au système d'exploitation pour lequel elles sont conçues ou nécessitent d'exécuter le système d'exploitation à l'intérieur d'un hyperviseur, ce qui affecte leur applicabilité dans une analyse forensique agnostique du système d'exploitation sur de vrais appareils.

Structure signatures and validation rules

Table 1: Structure signatures (○) and validation rules (●) for each MMU mode.

AArch64 (64 bit) Long MMU mode		
Object	**Type**	**Rule**
PTL0/1/2/3 Entry	○	Size(Entry) = 8 Bytes
Empty PTL0/1/2/3 Entry	○	Entry[0] = 0
PTL3 reserved entry 4/16/64Kib granule	○	Entry[0] = 1 ∧ Entry[1] = 0
PTL3 entry 4/64KiB granule	○	Entry[0,1] = 1 ∧ Entry[SH] ≠ 1
PTL3 entry 16KiB granule	○	Entry[0,1] = 1 ∧ Entry[SH] ≠ 1 ∧ Entry[12,13] = 0
Block PTL2 entry 4KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12…15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17…20] = 0
Block PTL2 entry 16KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12…15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17…24] = 0
Block PTL2 entry 64KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12…15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17…28] = 0
Block PTL1 entry 4KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12…15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17…29] = 0
PTL0/1/2 ptr 4KiB granule	○	Entry[0,1] = 1 ∧ Entry[Address] ∈ RAM
PTL0/1/2 ptr 16KiB granule	○	Entry[0,1] = 1 ∧ Entry[12,13] = 0 ∧ Entry[Address] ∈ RAM
PTL0/1/2 ptr 64KiB granule	○	Entry[0,1] = 1 ∧ Entry[12…15] = 0 ∧ Entry[Address] ∈ RAM
Kernel Radix tree	●	∃ DataPage | ReadOpcode(`ESR_EL1, FAR_EL1, ELR_EL1`) ∈ DataPage ∧ ∃ DataPage | WriteOpcode(`TTBR0_EL1, TCR_EL1`) ∈ DataPage ∧ ∃ DataPage | ExecOpcode(`ERET`) ∈ DataPage
Kernel Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[0,1]] = 0 ⇒ RadixT ∈ {Accepted Kernel radix trees}
User Radix tree	●	∃ DataPage | ExecOpcode(`RET`) ∈ DataPage ∧ ∃ DataPage | ExecOpcode(`BLR`) ∈ DataPage
User Radix tree	●[2]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[1]] = 1 ⇒ RadixT ∈ {Accepted User radix trees}
PTL0/1/2/3	○	Address(Table) alignment and Size(Table) compatible with `TCR_EL1` fields. See [Hol20] for more details.
TCR_EL1	○	TCR_EL1[6, 35, 59…63] = 0

Size(X) = The size of the object X.
Object[W,X,Y…Z] = Bit W,X and all the bits in [Y,Z] of Object.
Object[NAME] = Field 'NAME' of Object. See documentation for more details.
RAM = Physical address used to access to a system memory location (no MMIO, ROMs, etc.).
ReadOpcode(X,Y), WriteOpcode(X,Y) = Physical address of an opcode which reads/writes on register X or Y.
ExecOpcode(X) = Physical address of X opcode.
Address(X) = Physical address of the object X.
REGISTER[W,X,Y…Z] = Bits of the value contained in REGISTER.

[1] It exists at least a page writable in kernel mode and not writable in user mode.
[2] It exists at least a page readable or writable in in user mode.

AMD64 (64 bit) 4-level MMU mode		
Object	**Type**	**Rule**
IDT entry	●	Size(IDTEntry) = 16 Byte
IDT entry	●	Address(IDT) % 4 = 0
Empty IDT entry	●	IDTEntry[P] = 0
Used IDT entry	●	IDTEntry[P] = 1 ∧ IDTEntry[35...39,44,95:127] = 0 ∧ IDTEntry[TYPE] ∈ {14,15} ∧ IDTEntry[DPL] ∈ {0,3}
IDT table	●	Address(IDT) % 8 = 0 ∧ Size(IDT) = 4096 Byte
IDT table	●	∀ i ∈ {0...8,10...14,16...19} IDT[i][P] = 1 ∧ IDT[i][47...64] = 1
IDT table	●[1]	IDT[3][DPL] = 3 ∧ IDT[0,2,6,7,8,10...14][DPL] = 0
PT/PD/PDPT/PML4 Entry	○	Size(Entry) = 8 Bytes
Empty PT/PD/PDPT/PML4 Entry	○	Entry[P] = 0
PT entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0
PD 2MiB entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 1
PDPT 1GiB entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 1 ∧ Entry[21:29] = 0
PT/PD/PDPT pointer	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PD/PT/PDPT/PML4	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes
Radix Tree	●	∀ InterruptHandler ∈ {Found IDT} Resolve(RadixTree, InterruptHandler[Address]) = True

IDT[X][Y] = Field Y of the record X in interrupt table.
MAXPHYADDR = See documentation [Int20] for more details.
Resolve(R,V) = Resolve virtual address V using radix tree R returning True for success False otherwise.

[1] The DPL field of IDT entries defines the minimum CPU privilege mode which is allowed to execute the code pointed by the entry: 0 for kernel mode, 3 for user mode. We force the DPL field of some entries to be 0 because that entries are associated with interrupts which are managed by the kernel (e.g. NMI, invalid opcode, double fault etc.). Instead, we force the code pointed by the entry associated with breakpoint exception to have DPL = 3 because it is needed to permit to a user space process to debug other processes.

ARM (32 bits) Short MMU mode		
Object	**Type**	**Rule**
PTL1/PTL2 Entry	○	Size(Entry) = 4 Bytes
Empty PTL1/PTL2 Entry	○	Entry[0,1] = 0
Small PTL2 Page Entry	○	Entry[1] = 1 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Large PTL2 Page Entry	○	Entry[0] = 1 ∧ Entry[1] = 0 ∧ Entry[6...8] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Large PTL2 page entries group	○	if ∃ LPEntry in Table ⇒ ∃ LPEntry_j | $\text{Address}(\text{LPEntry}_{j+1}) = \text{Address}(\text{LPEntry}_j + 4) \wedge \text{Address}(\text{LPEntry}_0) \% 16 = 0)$ j ∈ {0...15}
Section PTL1 Entry	○	Entry[1] = 1 ∧ Entry[9,18] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Supersection PTL1 Entry	○	Entry[1,18] = 1 ∧ Entry[9] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Supersection PTL1 entries group	○	if ∃ SSEntry in Table ⇒ ∃ SSEntry_j | $\text{Address}(\text{SSEntry}_{j+1}) = \text{Address}(\text{SSEntry}_j + 4) \wedge \text{Address}(\text{SSEntry}_0) \% 16 = 0)$ j ∈ {0...15}
PTL2 pointer Entry	○	Entry[0] = 1 ∧ Entry[1] = 0 ∧ Entry[9] = 0
PTL1 Table	○	Address(Table) alignment and Size(Table) compatible with `TTBCR` fields. See [Int20] for more details.
PTL2 Table	○	Address(Table) % 1024 = 0 ∧ Size(Table) = 1024 Bytes
Kernel Radix tree	●	∃ DataPage | ReadOpcodes(`DFSR, IFSR`) ∈ DataPage ∧ ∃ DataPage | WriteOpcodes(`TTBR0, TTBCR`) ∈ DataPage
Kernel Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[PNX] = 0 ⇒ RadixT ∈ {Accepted Kernel radix trees}
User Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[NX] = 0 ⇒ RadixT ∈ {Accepted User radix trees}
User Radix tree	●[2]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[1]] = 1 ⇒ RadixT ∈ {Accepted User radix trees}
TTBCR	○	TTBCR[3] = 1 ∧ TTBCR[6, 31] = 0

[1] It exist at least a table entry executable i.e. it exist at least a page which contains code.
[2] It exist at least a table entry writable i.e. it exist at least a page which contains data.

MIPS32 (32 bit) TLB and Radix tree MMU modes		
Object	**Type**	**Rule**
Config	○	Config[4...6] = 0 ∧ Config[31] = 1
Config5	○	Config5[1,12,14...26] = 0
PageGrain	○	PageGrain[5...7,13...25] = 0
PageMask	○	PageMask[0...10,29...31] = 0
PWCtl	○	PWCtl[8...30] = 0
PWField	○	PWField[30...31] = 0 ∧ (if Config[10...12] ≥ 2 ⇒ ∀ i ∈ {GDI, UDI, MDI, PTI}PWField[i] < 12)
PWSize	○	PWSize[30...31] = 0 ∧ (if Config[10...12] ≥ 2 ⇒ PWSize[PTW] ≠ 1)
Wired	○	Wired[0...15] < Wired[16...31]

PowerPC (32 bit) SDR1+BAT MMU mode		
Object	**Type**	**Rule**
Hash Table Entry	○	Size(Entry) = 8 Bytes
Empty Hash Table entry	○	Entry[V] = 0
Used Hash Table entry	○	Entry[V] = 1 ∧ Entry[RPN] ≪ 12 ∈ RAM ∧ (Entry[R] = 1 ∨ Entry[C] = 0)
Hash Table entries group	○	HT[i][j] ≠ HT[i][k] i ∈ {0...Size(HT)/64 - 1} j,k ∈ {0...7} ∧ j ≠ k
Hash Table	○	Address(HT) % 16 = 0 ∧ Size(HT) ∈ {64Kib 128Kib 256KiB 512KiB 1MiB 2MiB 4Mib 8MiB 16MiB 32MiB}
Hash Table	○	∀ Entry ∈ HT | Entry[V] = 1 ⇒ Address(Entry)[16...31] = Address(HT)[25...31] ‖ Hash(Entry[VSID][10...18] ⊕ Entry[API]) ∧ (1 ≪ (Log(Size(HT) - 16) -1) ∨ (Address(HT) ≫ 16)[0...8]
Hash Table	●[1]	∃ Entry ∈ HT | Entry[V] = 1
Hash Table	●[2]	∃ Entry_1 and Entry_2 ∈ HT | Entry_1[V] = Entry_2[V] = 1 ∧ Entry_1[VSID] ≠ Entry_2[VSID]
Hash Table	●[3]	∃ Entry_1 and Entry_2 ∈ HT Entry_1[V] = Entry_2[V] = 1 ∧ Entry_1[RPN] ≠ Entry_2[RPN]
Parsed Hash Tables	●[4]	∀ j ∈ {Parsed HTs}|$H_j = H(\text{HT}_j) = -\sum P(\text{RPN}_i)\log_2 P(\text{RPN}_i)$; $H_{\max} = \text{Max}(H_j)$; if $H_j > 0.8H_{\max}$ ⇒ j ∈ {Accepted HTs}

HT[X][Y] = Field Y of the record X in hash table.

[1] It exists at least one not empty entry in the table.

[2] It exists entries associated with at least two different segments.

[3] The hash table maps for at least two different physical pages.

[4] The kernel, spreading data among a wide range of physical pages in RAM, increase the entropy of the distribution of the physical addresses of the pages in the real hash table. We filter the set of hash tables candidates discarding tables which have entropy less than the 80% of the maximum of entropy in the set.

RISC-V SV32 and SV48 MMU modes		
Object	**Type**	**Rule**
PTL0/PTL1 Entry (SV32)	○	Size(Entry) = 4 Bytes
PTL0/PTL1/PTL2 Entry (SV48)	○	Size(Entry) = 8 Bytes
Empty PTL0/PTL1(/PTL2) Entry	○	Entry[V] = 0
PTL0 entry (SV32)	○	Entry[V] = 1
PTL0 entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0
PTL1 4MiB entry (SV32)	○	Entry[V] = 1 ∧ Entry[10:19] = 0
PTL1 2MiB entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0 ∧ Entry[10:18] = 0
PTL2 1GiB entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0 ∧ Entry[10:28] = 0
PTL0 pointer (SV32)	○	Entry[V] = 1 ∧ Entry[R,W,X,D,A,U] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PTL0/PTL1 pointer (SV39)	○	Entry[V] = 1 ∧ Entry[R,W,X,D,A,U] = 0 ∧ Entry[54,63] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PTL0/PTL1(/PTL2)	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes

Intel x86 IA32 and PAE MMU modes		
Object	**Type**	**Rule**
IDT entry	●	Size(IDTEntry) = 8 Bytes
Empty IDT entry	●	IDTEntry[P] = 0
Used IDT entry	●	IDTEntry[P] = 1 ∧ IDTEntry[32:39,44] = 0 ∧ IDTEntry[42] = 1 ∧ IDTEntry[DPL] ∈ {0,3}
IDT	●	Address(IDT) % 4 = 0 ∧ Size(IDT) = 2048 Bytes
IDT	●	IDT[0...8,10...14][P] = 1
IDT	●[1]	IDT[0,2,6...8,10...14][DPL] = 0 ∧ IDT[3][DPL] = 3
PT/PD Entry (IA32)	○	Size(Entry) = 4 Bytes
PT/PD/PDPT Entry (PAE)	○	Size(Entry) = 8 Bytes
Empty PT/PD(/PDPT) Entry	○	Entry[P] = 0
PT entry (IA32)	○	Entry[P] = 1
PT entry (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0
PD 4MiB entry (IA32)	○	Entry[P] = 1 ∧ Entry[7] = 1 ∧ Entry[MAXPHYADDR...19, 21] = 0
PD 2MiB entry (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 1 ∧ Entry[13:20] = 0
PT pointer (IA32)	○	Entry[P] = 1 ∧ Entry[7] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PT pointer (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0 ∧ newline Entry[Address] ≪ 12 ∈ RAM
PD pointer (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0 ∧ newline Entry[1,2,5,6,8,63] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PD/PT	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes
PDPT (PAE)	○	Address(Table) % 32 = 0 ∧ Size(Table) = 32 Bytes
Radix Tree	●	∀ InterruptHandler ∈ {Found IDT} Resolve(RadixTree, Address(InterruptHandler))

[1] See Note 1 of AMD64 architecture.

References

[AD07] Ali Reza Arasteh and Mourad Debbabi. Forensic memory analysis: From stack and code to execution history. *Digital Investigation*, 4:114–125, 2007.

[avm23] Avml (acquire volatile memory for linux), 2023. URL: https://github.com/microsoft/avml.

[BA18] Manish Bhatt and Irfan Ahmed. Leveraging relocations in elf-binaries for linux kernel version identification. *Digital Investigation*, 26:S12–S20, 2018. URL: https://www.sciencedirect.com/science/article/pii/S1742287618302019, doi:10.1016/j.diin.2018.04.022.

[bar23] The Barrelfish OS, 2023. URL: http://www.barrelfish.org/.

[Ben23] Daryl Bennett. Lime - linux memory extractor, 2023. URL: https://github.com/504ensicsLabs/LiME.

[Ber18] Stefan Le Berre. From corrupted memory dump to rootkit detection, 2018. URL: https://exatrack.com/public/Memdump_NDH_2018.pdf.

[BGC11] Robert Beverly, Simson Garfinkel, and Greg Cardwell. Forensic carving of network packets and associated data structures. *Digital Investigation*, 8:S78–S89, 2011. The Proceedings of the Eleventh Annual DFRWS Conference. URL: https://www.sciencedirect.com/science/article/pii/S174228761100034X, doi:10.1016/j.diin.2011.05.010.

[bla23] BlackBag Technologies, 2023. URL: https://www.blackbagtech.com/.

[bui23] Buildroot, 2023. URL: https://buildroot.org/.

[Bur20] Jeffrey Burt. Alibaba on the bleeding edge of risc-v with xt910, 2020. URL: https://www.nextplatform.com/2020/08/21/alibaba-on-the-bleeding-edge-of-risc-v-with-xt910/.

[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? Technical report, DEFENCE RESEARCH AND DEVELOPMENT CANADA VALCARTIER (QUEBEC), 2011.

[CDP20] Zoran Cekerevac, Zdenek Dvorak, and Tamara Pecnik. Top seven iot operating systemsin mid-2020. *MEST Journal*, 8, 07 2020. doi:10.12709/mest.08.08.02.06.

[CGFB18] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*, pages 161–175, Los Alamitos, CA, USA, 2018. IEEE, IEEE Computer Society.

[CMMRI20] Andrew Case, Ryan D Maggio, Modhuparna Manna, and Golden G Richard III. Memory analysis of macos page queues. *Forensic Science International: Digital Investigation*, 33:301004, 2020.

[CMR10] Andrew Case, Lodovico Marziale, and Golden G. Richard. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010. The Proceedings of the Tenth Annual DFRWS Conference. URL: https://www.sciencedirect.com/science/article/pii/S1742287610000320, doi:10.1016/j.diin.2010.05.005.

[Coh14] Michael Cohen. Rekall memory forensic framework, 2014.

[com23] 9Front community. 9front os, 2023. URL: http://9front.org/.

[cpu23] `cpu_rec`, 2023. URL: https://github.com/airbus-seclab/cpu_rec.

[CRI17] Andrew Case and Golden G Richard III. Memory forensics: The path forward. *Digital investigation*, 20:23–33, 2017.

[CSN09] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[CSXK08] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 255–266, USA, 2008. USENIX Association.

[dar23] Darwin os, 2023. URL: https://github.com/apple/darwin-xnu.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[DGLZ+11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 297–312, USA, 2011. IEEE Computer Society. doi:10.1109/SP.2011.11.

[DGSTG09] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 566–577, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1653662.1653730.

[ea23a] Fabrice Bellard et al. QEMU - Generic and open source machine and user space emulator and virtualizer, 2023. URL: https://www.qemu.org/.

[ea23b] Fabrice Desclaux et al. Miasm - Reverse engineering framework, 2023. URL: https://github.com/cea-sec/miasm.

[ea23c] Sergi Alvarez et al. Radare2 - Libre and Portable Reverse Engineering Framework, 2023. URL: https://rada.re/n/.

[Edi23] OmniOS Community Edition. OmniOS, 2023. URL: https://omniosce.org/.

[emb23] Embox os, 2023. URL: https://www.embox.rocks/.

[FG21] Felix C Freiling and Michael Gruhn. Defining atomicity (and integrity) for snapshots of storage in forensic computing. In *Digital Forensic Research Workshop (DFRWS) Europe*, 2021.

[FHA+22] Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. Katana: Robust, automated, binary-only forensic analysis of linux memory snapshots. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 214–231, 2022.

[FL12] Yangchun Fu and Zhiqiang Lin. Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 586–600, USA, 2012. IEEE Computer Society. doi:10.1109/SP.2012.40.

[FPW+16] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 11–22, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897845.2897850.

[FPYL14] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th annual computer security applications conference*, pages 196–205, USA, 2014. Association for Computing Machinery.

[Fre05] FreeScale. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*. FreeScale, 2005.

[gen23] Genode os, 2023. URL: https://genode.org/.

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

[GFP+14] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing*, 2(4):380–394, 2014.

[Gib84] William Gibson. *Neuromancer*. Ace Science Fiction. Ace Books, July 1984.

[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*, CODASPY '16, page 62–72, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2857705.2857707.

[GLB13] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. Hypervisor Memory Forensics. In Salvatore J Stolfo, Angelos Stavrou, and Charles V Wright, editors, *Research in Attacks, Intrusions, and Defenses*, pages 21–40, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[hai23] Haiku os, 2023. URL: https://www.haiku-os.org/.

[HBHW07] Ewa Huebner, Derek Bem, Frans Henskens, and Mark Wallis. Persistent systems techniques in forensic acquisition of memory. *Digital Investigation*, 4(3-4):129–137, 2007.

[HBN09] Brian Hay, Matt Bishop, and Kara Nance. Live analysis: Progress and challenges. *IEEE Security & Privacy*, 7(2):30–37, 2009.

[Hef23] Craig Heffner. Binwalk, 2023. URL: https://github.com/ReFirmLabs/binwalk.

[hel23] Helenos, 2023. URL: http://www.helenos.org/.

[Hol18] ARM Holdings. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. ARM Holding, 2018.

[Hol20] ARM Holdings. *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. ARM Holdings, 2020.

[HSH+09] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[IBM17] IBM. *Power ISA. Version 3.0B*. IBM, 2017.

[Int20] Intel. *Intel 64 and IA-32 Architectures–Software Developer's Manual–Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. Intel Corporation, 2020.

[JM98] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[ker23] Complete virtual memory map with 4-level page tables, 2023. URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[Knu98] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. URL: https://www.worldcat.org/oclc/312898417.

[Kor07] Jesse D Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4(1):24–29, 2007.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

[Kre17] Kevin Krewell. Western digital gives a billion unit boost to open source risc-v cpu, 2017. URL: https://www.forbes.com/sites/tiriasresearch/2017/12/06/western-digital-gives-a-billion-unit-boost-to-open-source-risc-v-cpu/.

[LBF20] Tobias Latzo, Julian Brost, and Felix Freiling. Bmcleech: Introducing stealthy memory forensics to bmc. *Forensic Science International: Digital Investigation*, 32:300919, 2020.

[Lev15] Jamie Levy. Using PROT_NONE on Linux, 2015. URL: https://volatility-labs.blogspot.com/2015/05/using-mprotect-protnone-on-linux.html.

[LHKF21] Tobias Latzo, Florian Hantke, Lukas Kotschi, and Felix Freiling. Bringing forensic readiness to modern computer firmware. In *Digital Forensic Research Workshop (DFRWS) Europe*, 2021.

[LK08] Eugene Libster and Jesse D Kornblum. A proposal for an integrated memory acquisition mechanism. *ACM SIGOPS Operating Systems Review*, 42(3):14–20, 2008.

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

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

[LRW+12] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. Discovering semantic data of interest from un-mappable with confidence. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, USA, 2012. The Internet Society.

[LRZ+11] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2011. The Internet Society. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/lin.pdf.

[LSF21] Tobias Latzo, Matti Schulze, and Felix Freiling. Leveraging intel dci for memory forensics. In *DFRWS USA 2021*, 2021. URL: https://dfrws.org/presentation/leveraging-intel-dci-for-memory-forensics/.

[LZL21] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking "security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2937–2950, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3485253.

[LZLS19] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected {I/O} operations in {AMD's} secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019.

[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*, pages 1–18, USA, 2010. CERIAS - Purdue University.

[MC13] Andreas Moser and Michael I Cohen. Hunting in the enterprise: Forensic triage and incident response. *Digital Investigation*, 10(2):89–98, 2013.

[MCJ17] Daniel Mercier, Aziem Chawdhary, and Richard Jones. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501, USA, 2017. IEEE Computer Society, IEEE Computer Society.

[MFPC10] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings 13*, pages 297–316. Springer, 2010.

[min23] Minix3 OS, 2023. URL: https://www.minix3.org/.

[Mip15] Mips. *MIPS Architecture For Programmers Vol. III: MIPS32 / microMIPS32 Privileged Resource Architecture*. Imagination Technologies, 2015.

[mor23] Morphos, 2023. URL: https://www.morphos-team.net/.

[Nat23] National Security Agency. Ghidra - software reverse engineering framework, 2023.

[OB22] Andrea Oliveri and Davide Balzarotti. In the land of mmus: Multiarchitecture os-agnostic virtual memory forensics. *ACM Trans. Priv. Secur.*, mar 2022. Just Accepted. doi:10.1145/3528102.

[ODB23] Andrea Oliveri, Matteo Dell'Amico, and Davide Balzarotti. An os-agnostic approach to memory forensics. In *NDSS 2023, Network and Distributed System Security Symposium, 27 February-3 March 2023, San Diego, CA, USA*. Internet Society, 2023.

[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)*, pages 1751–1768, 2019.

[PB21] Fabio Pagani and Davide Balzarotti. Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security*, 25(1):1–26, 2021.

[PDB18] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 354–365, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3176258.3176306.

[PFB19] Fabio Pagani, Oleksii Fedorov, and Davide Balzarotti. Introducing the temporal dimension to memory forensics. *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–21, 2019.

[pow23] System Dump Facility, 2023. URL: https://www.ibm.com/docs/en/aix/7.2?topic=facilities-system-dump-facility.

[qnx23] QNX, 2023. URL: https://www.qnx.com.

[QQY22] Zhenxiao Qi, Yu Qu, and Heng Yin. LogicMEM: Automatic profile generation for binary-only memory forensics via logic inference. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022. doi:10.14722/ndss.2022.24324.

[RALJA21] Muhammad Haris Rais, Rima Asmar Awad, Juan Lopez Jr., and Irfan Ahmed. Jtag-based plc memory acquisition framework for industrial control systems. *Forensic Science International: Digital Investigation*, 37:301196, 2021.

[RAS14] Vassil Roussev, Irfan Ahmed, and Thomas Sires. Image-based kernel fingerprinting. *Digital Investigation*, 11:S13–S21, 2014. Fourteenth Annual DFRWS Conference. URL: https://www.sciencedirect.com/science/article/pii/S1742287614000565, doi:10.1016/j.diin.2014.05.013.

[ras23] Raspberry PI, 2023. URL: https://www.raspberrypi.org/.

[rco23] rCore, 2023. URL: https://github.com/rcore-os/rCore.

[rea23] ReactOS, 2023. URL: https://reactos.org/.

[red23] Redox OS, 2023. URL: https://www.redox-os.org/.

[RFP+12] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th annual computer security applications conference*, pages 79–88, 2012.

[RIC14] Golden G Richard III and Andrew Case. In lieu of swap: Analyzing compressed ram in mac os x and linux. *Digital Investigation*, 11:S3–S12, 2014.

[ris23] RISC OS Open, 2023. URL: https://www.riscosopen.org.

[SA19] O. Sardar and D. Andonov. White paper: Finding evil in windows 10 compressed memory. Technical report, FireEye, 601 McCarthy Blvd. Milpitas, CA 95035, 2019. URL: https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/finding-evil-in-windows-10-compressed-memory-wp.pdf.

[SB03] Peter F Sweeney and Michael Burke. Quantifying and evaluating the space overhead for alternative c++ memory layouts. *Software: Practice and Experience*, 33(7):595–636, 2003.

[Sch07] Bradley Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. *digital investigation*, 4:126–134, 2007.

[SG10] Karla Saur and Julian B. Grizzard. Locating x86 paging structures in memory images. *Digital Investigation*, 7(1-2):28–37, oct 2010. URL: https://linkinghub.elsevier.com/retrieve/pii/S174228761000054X, doi:10.1016/j.diin.2010.08.002.

[SSB10] Asia Slowinska, Traian Stancescu, and Herbert Bos. Dde: dynamic data structure excavation. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 13–18, New Delhi, India, 2010. USENIX Association.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2011. The Internet Society.

[Sto89] Clifford Stoll. *Cuckoo's Egg*. Doubleday Books, September 1989.

[SYLS18] Wei Song, Heng Yin, Chang Liu, and Dawn Song. 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*, CCS '18, pages 606–618, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243813.

[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*, pages 179–188, USA, 2010. IEEE Computer Society, IEEE Computer Society.

[tls23] TLSH - Trend Micro Locality Sensitive Hash, 2023. URL: https://github.com/trendmicro/tlsh.

[TQHMT21] Truong-An Tran-Quoc, Chi Huynh-Minh, and Anh-Duy Tran. Towards os-independent memory images analyzing: Using paging structures in memory forensics. In *2021 14th International Conference on Security of Information and Networks (SIN)*, volume 1, pages 1–8, 2021. doi:10.1109/SIN54109.2021.9699263.

[UGCL14] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. Sigpath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security*, pages 237–256, Cham, 2014. Springer, Springer International Publishing.

[vAv08] R.B. van Baar, W. Alink, and A.R. van Ballegooij. Forensic memory analysis: Files mapped in memory. *Digital Investigation*, 5:S52–S57, 2008. The Proceedings of the Eighth Annual DFRWS Conference. URL: https://www.sciencedirect.com/science/article/pii/S1742287608000327, doi:10.1016/j.diin.2008.05.014.

[VF12] Stefan Vömel and Felix C Freiling. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. *Digital Investigation*, 9(2):125–137, 2012.

[Vol23] Volexity. Volexity, 2023. URL: https://www.volexity.com/.

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

[VS19] Sebastian Vogl and Blaine Stancill. Rekall support for Windows 10 memory compression, 2019. URL: https://github.com/mandiant/win10_rekall/blob/win10_compressed_memory/rekall-core/rekall/plugins/windows/win10_memcompression.py.

[vxw23] vxWorks, 2023. URL: https://www.windriver.com/products/vxworks/.

[WA19] Asanovic K. Waterman A., editor. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, USA, 2019.

[Wal17] Aaron Walker. Volatility framework: Volatile memory artifact extraction utility framework, 2017.

[win23] Winpmem, 2023. URL: https://github.com/Velocidex/WinPmem.

[Won20] William G. Wong. Ada and risc-v secure nvidia's future, 2020. URL: https://www.electronicdesign.com/markets/automotive/article/21121197/ada-and-riscv-secure-nvidias-future.

[WRG15] James Wagner, Alexander Rasin, and Jonathan Grier. Database forensic analysis through internal structure carving. *Digital Investigation*, 14:S106–S115, 2015. The Proceedings of the Fifteenth Annual DFRWS Conference. URL: https://www.sciencedirect.com/science/article/pii/S1742287615000584, doi:10.1016/j.diin.2015.05.013.

[XLXJ12] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. Libvmi: A library for bridging the semantic gap between guest os and vmm. In *Proceedings of the 2012 IEEE 12th International Conference on Computer and Information Technology*, CIT '12, pages 549–556, USA, 2012. IEEE Computer Society. doi:10.1109/CIT.2012.119.

[xv623] XV6, 2023. URL: https://github.com/mit-pdos/xv6-riscv.