Technische Universität München
TUM School of Computation, Information and Technology

The Configurability of the Linux Kernel and the Implications on Security

Fabian Franzen

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr. Jens Grossklags
Prüfende der Dissertation:	1. Prof. Dr. Claudia Eckert 2. Prof. Dr. Davide Balzarotti

Die Dissertation wurde am 26.06.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 24.10.2024 angenommen.

Acknowledgements

I would like to use the opportunity to thank all the great people who accompanied me on my research career and without this dissertation would not have been possible:

- My family and friends. Thank you very much for your constant support through all the ups and downs on the way to my doctorate.
- My advisor Prof. Claudia Eckert who catched the interest my interest in IT security in her lectures and, finally, gave me the opportunity to work in her research group and on my research topic.
- Prof. Jens Grossklags for mentoring me and for his steady advice on how to write good research papers.
- Prof. Balzarotti for becoming my second examiner.
- My great colleagues (Lukas, Fabian, Maximilian, Ludwig, and Manuel) who are a constant source of moral support and are always open to discuss the next crazy research idea.
- My former colleagues Julian, Lorenz, Clemens, Bruno and Tom, who deeply impressed and inspired me with their skills during the first CTFs we played together.
- My former (and current) students Tobias Holl, Lion Steger, Andreas Chris Wilhelmer, Josef Schönberger, and Sebastian Kappes, who invented papers with me and proofread stuff in several nightshifts.
- All current and previous ITSec tutors who made teaching at university so much fun!

Abstract

In today's world, the amount of threat actors with access to sophisticated tools and methods has grown. They attack a wide range of IT, from simple personal computers to critical IT infrastructure. For some threat actors, attacking systems is merely a business model to generate profit. Others, which could include nations, use attacks as a potent tool to sway the outcomes of modern global conflicts in their favor. Despite years of research for sophisticated defenses and hardening mechanisms, the sad reality is that some IT systems will eventually be compromised in these attacks. Despite even better protection mechanisms, tools to analyze sophisticated attacks in a post-mortem scenario and ways to deploy new defenses faster are also required.

Therefore, this dissertation takes a multifaceted approach, focusing on three core areas: Efficient memory forensics on Linux systems, approaches to utilize the in-depth system knowledge in forensics analysis tools to add new security features dynamically to existing systems, and the strategic use of honeypots to better understand threat actor behavior.

To improve memory forensics, we developed a novel tool, *Katana*, aiming to transcend the limitations of existing approaches by offering configuration-agnostic memory forensics capabilities to analyists. The effectiveness of *Katana* is evaluated through an empirical analysis on all popular Linux distributions. *Katana* demonstrated in all cases that it can extract important information from the system where traditional approaches fail. We also developed an anti-forensic framework named *RandCompile* to test the limits of *Katana*. This framework employs the methods of *Software Diversity* and can, if all features are enabled, significantly reduce the effectiveness of *Katana*, but not fully prevent it.

Furthermore, we study how runtime information and hot patching features of the Linux operating system, cannot only assist forensic analysis but also be used to hotpatch a new security feature into the Linux kernel. More specifically, we invented a Linux kernel module capable of capturing a memory snapshot to enable a forensic analysis. Furthermore, we developed *FridgeLock* to showcase how an exemplary security feature—*Suspend Time Memory Encryption*—could be added without recompiling the kernel.

Last, we study honeypots and how common they are on the internet by performing an internet-wide scan. A comprehensive analysis is conducted to assess the significance of stealthiness in effectively attracting hackers. We have been able to find known open-source honeypots on the public internet as well as unknown ones. Our validation showed that our methods classify a honeypots correctly in more than 99 percent of the cases.

Zusammenfassung

In der heutigen Welt professionalisieren Angreifer ihre Werkzeuge und Tools, um eine möglichst breite Palette von (kritischen) IT-Infrastrukturen anzugreifen. Für einige Angreifer ist dies nur ein lukratives Geschäftsmodell, während es für andere (z.B. Staaten) sogar zu einem Werkzeug geworden ist, um die modernen Konflikte unserer Welt zu ihren Gunsten zu beeinflussen. Trotz jahrelanger Forschung nach ausgeklügelten Verteidigungs- und Härtungsmechanismen, werden auch heutzutage immer wieder IT-Systeme kompromittiert. Es ist daher notwendig noch bessere Werkzeuge zur Analyse und Beweissicherung nach einem Angriff (Post-Mortem-Szenario) und neue Mechanismen zur schnellen Bereitstellung neuer Abwehrmaßnahmen zur Verfügung zu haben.

Diese Dissertation konzentriert sich auf drei Bereiche: Effiziente Memory-Forensik auf Linux-Systemen, Nutzung des Systemwissens der Memory-Forensik, um bestehenden Systemen dynamisch neue Sicherheitsfunktionen hinzuzufügen, und die Untersuchung des Angreiferverhaltens mithilfe von Honeypots. Um die Memory-Forensik zu verbessern, haben wir *Katana* entwickelt, dass im Gegensatz zu bestehenden Tools eine Analyse eines Linux-Systems ohne Kenntnis der exakte Build-Umgebung vornehmen kann. Die Wirksamkeit von *Katana* wurde durch eine empirische Analyse von Speicherabzügen aller gängigen Linux-Distributionen getestet. *Katana* hat in allen Fällen gezeigt, dass es die für eine forensisch Analyse relevanten Informationen aus einem System extrahieren kann, wo traditionelle Ansätze versagen. Zusätzlich haben wir ein Anti-Forensik-Framework namens *RandCompile* entwickelt, um die Grenzen des von *Katana* verfolgten Ansatzes zu testen. Dieses Framework nutzt *Software Diversity* und kann, in der stärksten Ausbaustufe, die Wirksamkeit von *Katana* und anderen forensischen Frameworks erheblich reduzieren.

Darüber hinaus untersuchen wir, wie Laufzeitinformationen und Hot-Patching-Funktionen des Linux-Betriebssystems nicht nur die forensische Analyse unterstützen können, sondern auch zum dynamischen Hinzufügen einer neuen Sicherheitsfunktion in den Linux-Kernel verwendet werden können. Wir haben hierfür ein Linux Kernel Modul programmiert, dass ohne die Build-Umgebung des Wirtskernels Speicherabbilder erstellen kann. In einem zweiten Projekt haben wir *FridgeLock* entwickelt, um zu zeigen, wie eine beispielhafte Sicherheitsfunktion – *Suspend Time Memory Encryption* – zu einem System nachträglich hinzugefügt werden kann.

Abschließend untersuchen wir Honeypots und ihre Häufigkeit im Internet durch einen internetweiten Scan. Dessen Ergebnisse wurden auf bekannte und unbekannte Honeypots hin untersucht. Wir konnten bekannte Open-Source-Honeypots sowie neue, bisher Unbekannte finden. Unsere Validierung zeigte eine korrekte Klassifizerung in mehr als 99 Prozent der Fälle.

Contents

Publications

Ordered by date of publication

Martin Geier, **Fabian Franzen**, and Samarjit Chakraborty. Hardware-accelerated data acquisition and authentication for high-speed video streams on future heterogeneous automotive processing platforms. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2018.

Tobias Holl, Philipp Klocke, **Fabian Franzen**, and Julian Kirsch. Kernel-assisted debugging of linux applications. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, pages 1–9, 2018.

Fabian Franzen, Manuel Andreas, and Manuel Huber. FridgeLock: Preventing data theft on suspended linux with usable memory encryption. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 215–219, 2020.

Maximilian von Tschirschnitz, Ludwig Peuckert, **Fabian Franzen**, and Jens Grossklags. Method confusion attack on bluetooth pairing. In *2021 IEEE symposium on security and privacy (SP)*, pages 1332–1347. IEEE, 2021.

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.

Fabian Franzen, Lion Steger, Johannes Zirngibl, and Patrick Sattler. Looking for honey once again: Detecting rdp and smb honeypots on the internet. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 266–277. IEEE, 2022.

Fabian Franzen, Andreas Chris Wilhelmer, and Jens Grossklags. Randcompile: Removing forensic gadgets from the linux kernel to combat its analysis. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 677–690, 2023.

Chapter 1

Introduction

Computer systems have become an ubiquitous part of our society. However, with the increase of digitalization, our society's dependence on information technology has reached a level where its stability has become crucial. This has lately become apparent again with the escalation of the Russian-Ukrainian war and the war between Hamas and Israel.

The Google Threat Analysis Group (TAG), which is responsible for countering threats to Google services and the overall internet, has recently published two reports about the nature and the impact of these armed conflicts [1], [2].

Since 2012, Iran is known to attack key organizations in the middle-east including Israel. With the start of the Hamas the attacks on Israel have been intensified. Also cyber attacks on institutions in the US and Albanina (a NATO member state) have occurred. While not all attacks can be correctly attributed, there is a general increase of cyber attacks for espionage, desinformation to shift the public opinion pro Hamas and disruption attacks with wiper malware. Likewise, Israel seems to have attacked Iran and caused disruptions to payment systems in several Iranian gas-stations [1].

While cyber-espionage is known to be a problem for years, Google TAG stated in 2023 in the context of the Russian-Ukrainian war: "Importantly, this marks the first time that cyber operations have played such a prominent role in a world conflict". During the first four months of 2022 (the begin of the conflict), Mandiant (part of Google) has observed more destructive cyberattacks in Ukraine than in the previous eight years. One notable example, is the attack to Viasat KA-SATs broadcasting service shortly before Russia's invasion of Ukraine, which affected not only Ukraine but also thousands of Enercon's wind turbines in Germany [2].

Furthermore, the Federal Ministry for Information Security in Germany (BSI) reported that ransomware is still the predominant malware in 2023 [3]. Many threat actors in this field have matured in professionalism and occasionally do not even attack companies themselves anymore but sell their tools to other groups willing to perform the actual attacks. Despite all the security research conducted in the past, the number of weaknesses and critical vulnerabilities discovered in software is still high [3].

Although effective security concepts, potential attack vectors, and corresponding defense mechanisms have been researched for decades, the methods for analyzing attacks and threat actors must become even more effective to keep up with the emerging threat landscape. In the case of ransomware, a bug in the ransomeware itself might recover the encryption key used and allows for a recovery of the data without paying the ransom. Furthermore, an effective way to

analyze a past attack might be the only way to assign it to a known group or state. As these attributions can have severe legal or political consequences, it is crucial to do this correctly and effectively. Even if it is impossible to attribute an attack to a threat actor, an attack analysis will show weaknesses in the current system and maybe new attack patterns that system operators can use to detect future attacks in Intrusion Detection Systems.

Therefore, this theses tries to tackle two different topics out of the area of threat analysis research. The first topic is *Effective Linux Kernel Forensics*, which aims at easing the analysis after an attack to a Linux-based system has occurred. While the field of forensics on Linux systems has received a considerable amount of research from 2003 to 2014, the advances have not transitioned into operational tools. This indicates unresolved issues that need to get resolved. Furthermore, a forensic analysis, which will reveal the needed information about the steps an attacker took into the system, requires a deep knowledge of the inner workings of the Linux kernel. Otherwise, an analyst cannot make sense out of the data his tools collect. As we will see, the runtime data of the Linux operating system, which enables forensic analysis, can also be used to add security functionality to an already deployed system in an automated manner. To showcase this method, we developed *FridgeLock*, which adds *Suspend Time Memory Encryption* into an already compiled kernel. This capability allows developers to design security features for existing systems without the need to update them. In the case of a legacy system, this might be the only effective way to deploy such a feature. Details are outlined in Chapter 2.

This thesis furthermore analyzes the honeypot ecosystem in the public IPv4 address space of the Internet. *Honeypots* are used to attract hackers and study their methods and have, therefore, a similar purpose as a forensic analysis. *Honeypots* have proven to be a valuable tool to gain insights about ongoing attacks, which is vital, as stated above. As they are specially prepared systems, the forensic analysis of break-in attempts is more straightforward than the analysis of previously unknown ones. The systems usually keep dedicated log files and network packet dumps that allow for a detailed step-by-step recovery of the path of an attacker. However, the need for stealthiness requires the use of Virtual Machine Introspection tools, which function similarly to forensic analysis tools. While literature reports them to be a useful tool in practice, we realized that several of the openly available tools have become outdated and that the malware samples that we where able to obtain with them are faulty. Therefore, we conducted an internet-wide scan of the overall IPv4 address space for *honeypots* to detect these implementations and to measure how widespread they are in use. Furthermore, our scanner can also detect protocol implementations that emulate popular network protocols closely but not perfectly, which could indicate the presence of an unknown high *honeypot* implementation. Further, we study how the stealthiness of a honeypot influences its chance of being attacked. Our data offers insights into the mutual relationship between server administrators and attackers on the Internet. The details are outlined in Chapter 3.

Chapter 2

Topic 1: Effective Linux Kernel Forensics

While forensics, in general, is the art of reconstructing events at a crime scene and holds high importance, digital forensics plays a crucial role in modern cybersecurity and law enforcement by investigating digital devices and uncovering evidence. In Chapter 1, we have already described a few of the new challenges that arose in the past years and why tools being able to reconstruct the series of events after a security incident are in high demand. In this chapter, we will delve into the challenges of *Memory Forensics*, why *Memory Forensics* is of particular interest even though it has received a significant amount of research in the past, and the unique challenges that exist on the Linux operating system.

Linux is a major, widely used open source operating system and, therefore, also of central interest to the security research community. The Android operating system for smartphones is based upon the Linux kernel with a few additional patches. According to Statista, Android has a market share of about 70 percent among mobile operating systems at the end of 2023 [4]. Likewise, Linux is very popular in cloud scenarios. According to Microsoft Azure, more than 60 percent of their customers are using Linux [5].

A key advantage of Linux is its openness, which allows various communities to study the inner workings of the operating system, experiment with them, and add additional features as needed. Hardware and platform developers can develop and test their systems for the Linux kernel and share drivers and successful new features back with the community for the integration in the mainline kernel. However, not every feature is necessary for every use case and passes the bar for integration into the Linux mainline branch. The open nature of Linux renders this less of a problem for the vendor if he is willing to spend the effort on maintaining an out-of-tree patch for the kernel. He can still use the patch on his own devices or offer it to interested users for download and installation.

Linux is used on very memory constrained IoT devices with low CPU and memory resources as well as on systems with multiple CPUs and terabytes of main memory. To allow for all these different use cases, the Linux kernel developers have invented a modular configuration system. The build configuration of a kernel describes which specific implementation should be chosen for the kernel to be built (see Section 2.5 for details). To allow even further flexibility, the Linux kernel allows for dynamic loading of modules at runtime that can be built without the need to compile the whole kernel. This allows every system developer to create an optimal kernel for his (potentially very narrow and specific) use case.

2.1 Memory Forensics

Digital Forensics covers analysis of security incidents of an IT system, like a classic forensic investigation at a crime scene. During a forensic analysis, usually the following questions are asked: What happened on the system? Who performed which actions? What data got leaked? This information can be restored from logs of the operating system or application specific tools, temporary files created during processing of data, or firewall logs.

Memory Forensic is a sub-branch of Digital Forensics focusing on the volatile random access memory (RAM). While digital evidence can also be collected from disks, some data usually exists only in the RAM of the system under investigation, e.g. disk encryption keys, memory-resident code fragements, personal communication via e-mail or modern instant messaging solutions like Signal if they do not write a history to disk. However, this information might still be present a long time in the system, even after the termination of the respective process. In a case of a carefully crafted attack, attackers might overlook traces they have left in system memory. This makes Memory Forensics, which focuses on the main memory instead of artifacts found on disk, an interesting field of research [6] and in some cases the only tool left to obtain information about a past attack.

When analyzing a system, the first step is to take a snapshot of the memory, which can be archived as digital evidence and then analyzed by multiple tools without the need for the original device. As the operating system is a central building block of every computer system, which must hold information about running processes and open network connections, it is also a central source of information for the forensic analysis tools. Even if the analyst is only interested in the specific data of a selected process, the help of the operating system data structures might be needed, because they contain for each and every process the memory layout. As the memory snapshot may just be a copy of the physical RAM, no alternate information source to obtain a mapping between virtual and physical addresses might be available.

Simple data, like the kernel's log messages, can often be extracted by a simple search for ASCII printable strings in the memory dump. For more complex tasks, a dedicated forensic tool is advisable. A popular framework for forensic analysis of a system's memory snapshot is the *Volatility* framework [7]. *Volatility* is written in Python, and the different analysis methods are organized into separate plugins. The plugins rely heavily on the deterministic layout of the operating system's data structures which the framework provides to them through the use of debugging information of the operating system under analysis. In 2011, Google forked the *Volatility* project to reorganize the codebase and named the fork *Rekall Forensic* [8]. However, the project was abandoned in 2020, and active development seems to continue in the original project.

To get a better understanding of what information can potentially be extracted from system RAM, we will describe the different phases of the Linux kernel boot process and their influence on the information which are stored in the system RAM in the following section.

2.2 The Linux Kernel Boot Process

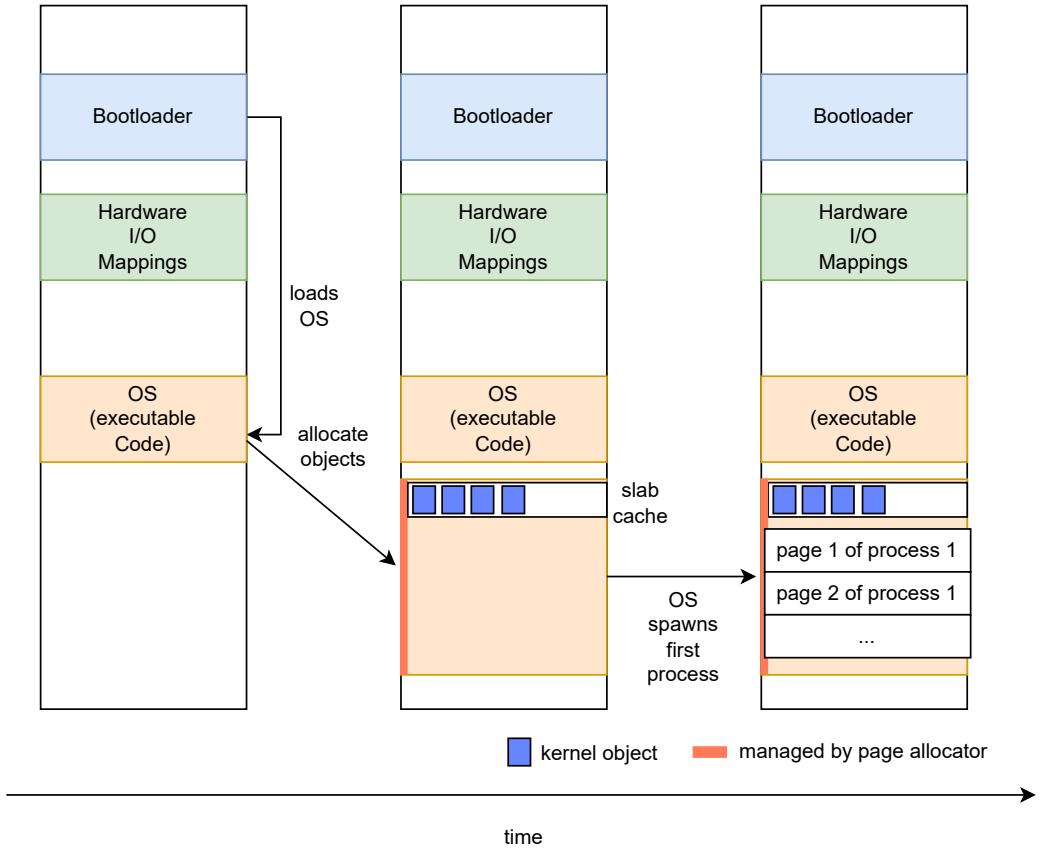

Figure 2.1: System memory in the different phases of boot

The boot process involves many steps and differs depending on CPU architecture and kernel version involved. Therefore, our description can only be an incomplete description highlighting the most relevant parts for this thesis. Figure 2.1 shows the memory contents of a off-the-shelve Linux system from system startup to the start of the first user space process. At startup, the system memory contains just the boot code of the hardware manufacturer and the bootloader. Furthermore, depending on the architecture, additional memory mapped regions might be present to allow the operating system to interface with the present hardware. By writing or reading to predefined locations, data can be exchanged between the CPU and the hardware. At this early stage during boot, the bootloader is in charge of loading the operating system into main memory. This might include decompressing it, checking its cryptographic signature in case of a secure boot environment, before it finally hands over the execution to it.

In case of the Linux operating system, the kernel will continue this boot process. Traditionally, literature describes this step as the operating system querying the hardware or its firmware (i.e. the BIOS) and configuring it to its needs [9]. However, there are also some security relevant concepts that come into play during boot that are less well known. First of all, there is Kernel Address Space Layout Randomization (KASLR), which is as concept explained in further detail in Section 2.3. While this feature can be disabled on a Linux system, it is enabled by default. When enabled, the kernel is relocated to a random position in main memory as one of the first steps after the control has been transferred to the OS. For the randomization, the kernel tries to obtain some entropy from the hardware to seed its cryptographically secure random number generator. As next step in the boot process, the hardware is set up, e.g. the different cores of the system, networking hardware and other input/output devices. Furthermore several runtime data structures are allocated like dedicated heaps for dynamical allocation of kernel objects. Examples for these dynamic kernel objects are information objects about the running processes, the files currently opened by them, received network data and many more. These data structures are allocated as they are needed during the trace of execution and usually *not* erased when they are freed. Instead, they are cleared when they are handed over to a not trustworthy userspace process. To manage available memory, the kernel has multiple memory management systems that are layered on top of each other. The SLAB allocator manages small sized kernel objects and requests the needed memory from a *page allocator* which manages whole physical pages of memory.

When all these steps are completed, the Linux kernel starts the init process (PID 1) as first process of the system, which in order starts various child processes that create the login prompt or offer other services to the user or the network. The memory for the userspace process is requested from the *page allocator* as needed.

On many systems, the boot loader places an initial ramdisk next to the kernel in system RAM from which the init process (PID 1) is started. The ramdisk contains Linux Kernel Modules needed during early boot which are loaded by the init process and `modprobe` tool. This dynamic approach allows kernel drivers not to be compiled into the kernel, saving memory if they are not needed during operation. This could be the case, if the hardware, the specific file system implementation or another Linux feature is not present or needed on the current installation. Optionally, the kernel can verify the integrity of the loaded modules by checking if cryptographic signatures of the modules are correct and signed by the respective Linux distributor. Furthermore, additional software to prompt the user for the disk encryption key is run if full disk encryption is enabled, before proceeding. Eventually, after all necessary modules have been inserted, the early init process mounts the real root file system into the kernel and runs a real init process like systemd.

It should be noted again, that memory is in general *not* securely erased when it is not needed anymore, but only when it is given to a non trustworthy instance[1]. Therefore, fragments of the

[1] There are exceptions to this rule. I.e. the `.init` segment is unloaded during boot

boot loader could still be in memory after the second and third userspace processes have been spawned, assuming the system has enough available memory. As a result, a malware infection at boot time could potentially be reconstructed for a long time by forensic analysis.

2.3 Address Space Layout Randomization

While the potential of information that could be excavated using memory forensics is huge, several security mechanisms of today's systems modify a deterministic memory layout and complicate the process of memory forensics. We will describe these in the following.

Address Space Layout Randomization (ASLR) for userspace programs and its counterpart for the operating system Kernel Address Space Layout Randomization (KASLR) are exploit mitigation mechanisms in software. The idea of Address Space Layout Randomization is to add some diversity into a software system, so that a wide-spread piece of software can not be attacked with a single exploit using the exact same steps for every system [10]. In the case of ASLR, the code and data of userspace programs will be loaded to *random* addresses in virtual memory on program startup instead of *deterministic* ones. Because of that, every executed program will (depending on the entropy of the randomization) have a slightly different memory layout. For Linux, the PAX team, which has proposed numerous security improvements for Linux, suggested and discussed ASLR as early as 2002 [11]. KASLR follows the same idea: During the early boot stages, the location of operating systems executable code and data is copied to a new randomized position and the old version is abandoned. In contrast to userspace ASLR, the operating system can randomize its position in physical and virtual memory, to harden it to attacks from hardware or its drivers which also deal with physical addresses. For Linux, the position in the physical memory and its position in virtual memory are both randomized.

This randomization mitigates binary exploitation attacks in many cases, because these usually require knowledge of pointer values to existing code fragments or data structures. For example, in Return Oriented Programming (ROP), an attacker will use existing code snippets (from the OS/userspace program) and puzzle them together to create a new malicious program. The puzzling is usually done beforehand by the attacker and coded into the exploit. However, if the concrete pointers (memory addresses) of the code fragments are not available, the attack cannot be performed or the exploit must obtain these addresses from other sources prior to the attack. In a non-ASLR scenario, an attacker can extract the necessary pointers on his own machine if he obtains a copy of the software. On most systems, software products are compiled only once by the software vendor (or distributor) and then simply copied – without modification – to all machines. This is the case for all popular operating systems such as Windows, Linux and MacOS.

The effectiveness of (K)ASLR depends on the secrecy of the chosen memory locations. If an attacker can guess or leak them from the running system, he can defeat the defense. This has led to discussions in the past about whether KASLR can provide an effective defense for the

Linux operating system, as it had several bugs that resulted in the leakage of memory addresses to userspace applications. In some cases, memory addresses have also been intentionally used in kernel system logs for debugging purposes. The situation is made worse by the fact that the Linux kernel is only randomized once at boot time and can have a fairly long uptime, giving an attacker a larger window to guess a valid address [12]. Nevertheless, KASLR and ASLR are now used on all major operating systems and have developed into a proven defense mechanism.

2.4 Structure Layout Randomization

As ASLR only works against attacks that rely on the knowledge of memory locations such as ROP, it is ineffictive against memory corruption attacks that work without. Consider the C program snippet in Listing 2.1, which shows the implementation of a `create_non_root_user` function. The function create a `user` object on the program heap, intializes it with no permissions and reads in the name of the new user from the terminal. It contains a buffer overflow in line 16, as the `fgets` is allowed to read more bytes into the name field than it can contain. In this setting an attacker can easily craft a username that overflows into the `is_admin` field of the user struct and that will be evaluated to `true` later on in the program (this part is not shown here).

```
1  #include <stdlib.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4
5  struct user {
6    char name[16];
7    bool is_admin;
8  };
9
10 typedef struct user* userp;
11
12 userp create_non_root_user() {
13   userp new_user = malloc(sizeof(struct user));
14   new_user->is_admin = 0;
15   // Buffer Overflow. is_admin is only 16 bytes long, but reading 255
       from terminal.
16   fgets(&new_user->name, 255, stdin);
17   return new_user;
18 }
```

Listing 2.1: Buffer-Overflow inside a structure

This exploit would not be affected by the use of (K)ASLR, as it does not require the use of a single memory address. However, the potential impact of the attack could be quite severe.

While it might be questioned, how often this example occurs in real world programs, there are similar scenarios where (K)ASLR does not offer effective protection.

This has led to the development of *Structure Layout Randomization*, which randomizes the data layout of structures in C. In contrast to (K)ASLR, it is not in wide-spread use and is only used as an optional feature inside the Linux kernel to mitigate binary exploitation attempts. It allows a developer to mark critical data structure with a special compiler attribute, which causes the compiler to reshuffle the fields of the structure at compile time. In the example above, this would lead to two possible outcomes: The structure layout is not changed and the program remains exploitable *or* the layout is reversed, which would result in a program that is *not* exploitable[2] in the way sketched above. The defense becomes more effective with larger data structures, because more variations become possible. At least, like as (K)ASLR, this mitigation results in an increase of software diversity and, therefore, decreases the chance of single easy exploit for all deployments of the software.

Unfortunately, this mitigation has a severe drawback that limits its effectiveness in practice: It is applied at compile time, which means that a security-aware user needs to compile their own Linux Kernels for the machines under his control. If the binary artifact becomes public, Structure Layout Randomization will – like (K)ASLR when addresses are leaked – become much less effective.

While this feature was primarly meant as a binary exploitation defense, it also limits the use of forensic tools that investigate the contents of the OS memory to reconstruct series of the events. These tools are further discussed in Section 2.1.

2.5 Linux Kernel Configurability

As mentioned, the Linux kernel is built using a flexible configuration system (KConfig). KConfig gives the user high flexibility in choosing which components the compiler should include in the build of the final kernel. Depending on the target CPU architecture and platform, different device drivers, debugging features, scheduling algorithms, and security features might be required. For example, a resource-constrained IoT device might not need support for the symmetric multiprocessing (SMP) feature if it contains only a single CPU core.

Depending on the features enabled, the number of functions and compilation units included in the compiled kernel changes, resulting in a smaller or larger kernel. It also changes the layout of the kernel's core data structures. Let's consider the feature for symmetric multiprocessing (SMP) and its impact on the data structure that represents a task in Linux (the `task_struct`). A simplified definition of it is depicted in Listing 2.2.

If the kernel is built with the feature `CONFIG_SMP`, the data structure has 72 bytes in size and the linked list `tasks`, which form the linked list between all running processes, is located at an

[2] It should be noted that the buffer overflow on the heap might result in other exploitation scenarios, but these will most likely necessitate the use of memory addresses during exploitation

```
struct task_struct {
  /* Per task flags (PF_*), defined further below: */
  unsigned int          flags;          /* offset 0 */

#ifdef CONFIG_SMP
  /* CPU the process is running on */
  int                   on_cpu;         /* offset 4 */
#endif

  /* Linked list of all running tasks */
  struct list_head      tasks;          /* offset 8; size 16 bytes */
#ifdef CONFIG_SMP
    /* Load average */
  int                   avg;            /* offset 24 */
#endif

  /* Description of virtual address space */
  struct mm_struct      *mm;            /* offset 32 */
  /* Process IDentifier */
  pid_t                 pid;            /* offset 40 */
  /* Security privileges of the current process */
  const struct cred __rcu   *cred;  /* offset 48 */
  /* The process name */
  char                  comm[16];       /* offset 56 */
}; /* size 72 bytes */
```

Listing 2.2: The `task_struct` of the Linux 6.9 kernel (simplified for clarity)

eight byte offset relative to the start of the structure. In the other case, where a user disables this feature, the `CONFIG_SMP` feature, the preprocessor will remove the fields `on_cpu` and `avg` from the structure. To fill the gaps in the memory layout, the compiler will move the memory location of the remaining structure fields towards the beginning.

The challenges of high configurability of the Linux kernel has already been subject to academic research. In 2014, Tartler *et al.* [13] found the Linux 3.2 kernel to have more than 12,000 configurable features which control over 89,000 `#ifdef` blocks. This causes a numerous problems: First, the parts of the Linux kernel which are not enabled are not even compiled and therefore not checked for syntax errors. Second, a static program analysis of the Linux kernel becomes even harder to perform. While analysis tools can barely analyze a single configuration of the Linux kernel, the complete analysis of all variants of such a huge software project is infeasible. The kernel developers try to mitigate the first problem by cautiously auditing new patches and generating various random kernel configurations (generated by `make randconfig`) in their continuous integration system [14]. However, this probabilistic approach seem to miss errors. The *Vampyr* tool managed to find hundreds of configuration dependent bugs [13] in the kernel.

Vampyr functions by analyzing the dependencies of the configurable features and generating a small set of distinct configurations which have a offer a maximal coverage to the analysis tool.

Besides its impact on analysis tools, drastically data layout changes through small changes of configuration and code have also implications on the dynamic module system of Linux. This dynamic module system allows the Linux kernel to load drivers and optional functionality as *Linux Kernel Module (LKM)* from disk. These LKMs can be build independently and can be loaded into the kernel when needed. However, an configuration might not be available, if the kernel has become outdated or is build by a hardware device vendor who does not share its configuration.

2.6 Linux's Runtime Information

Besides features that add randomness to the kernel, there are also features that allow for reasoning about its structure. First, the Linux Kernel maintains a table of its *exported functions* with the corresponding addresses in memory. This table is needed during the load of Linux Kernel Modules to integrate the module with the rest of the kernel.

Furthermore, the Linux Kernel can contain information symbol names and addresses of all functions that are still visible in the final build step. This feature is named *Kallsyms* and allows the kernel to offer augmented backtraces (in case of an error), live patching of security vulnerabilities, and tracing features (e.g., ftrace) to the user. This feature is almost always enabled in Linux systems and has existed since version 2.6.4 (released in 2004). While the storage format of the *Kallsyms* information has changed a few times over the last decades, it still can be used as a vantage point for further analysis (more details can be found in the core publication in Section 5.2)

2.7 Challenges

As depicted in Section 2.1, major forensic tools like *Volatility* and *Rekall Forensic* require the *exact* debugging information to perform their analysis tasks. However, a change in the Linux kernel configuration or the activation of Structure Layout Randomization will render existing debugging symbols outdated. Even worse, the debugging information of the Linux kernel does not soley depend to its configuration but also to its exact version of the kernel to analyze. However, this complicates forensic analysis in practice: A small change in the kernel configuration or a bugfix which includes a change to a core data structure could cause the forensic tool to malfunction. While this problem could theoretically be solved by a (central) authority that maintains an archive of different major Linux kernels used in production, this approach has failed. The Volatility foundation used to run a central GitHub Repo collecting such information. While this repository has only tried to cover major and well-known Linux distributions like *Ubuntu*, *Red-Hat* and *OpenSuse*, this repository has become outdated.

Even if this central repository approach would work, an analyst would still be unable to perform a forensic analysis of a Linux-based IoT device running a customized kernel of it's vendor.

This leads us to our fist research question:

> **Research Question 1**
> Can memory forensic tools be built which do not need to know the exact Linux kernel configuration and which would even work in the presence of Structure Layout Randomization (or other defense mechanisms based on *Software Diversity*)?

To solve this problem, we will investigate if Linux's own runtime information (see Section 2.6) are suitable vantage points for a deeper analysis.

As described in Section 2.5, not only forensic analysis is affected by the configurability of the kernel, but also the creation of LKMs that allow the user to extend the kernel with new security features. To update old systems as well as new systems, it would be beneficial if we could construct a LKM which loads into an existing kernel without knowing its configuration. Such an LKM could i.e. add security features to an device which hardware vendor does not support anymore. Furthermore, the Linux kernel developers marked not all functions of the kernel usable for the use by LKMs and seem to have focused on functions for device drivers. Some kernel functions, which would be crucial to implement a post-deployment security solution, are however not exported.

However, if we can solve RQ1, we gain very precise knowledge about the inner functions of the operating system under analysis. Therefore, using configuration-agnostic memory forensic methods could enable the development of security solutions for such systems, which we will try to showcase.

> **Research Question 2**
> How easily can we add a security feature to a deployed (preferably without knowing its configuration), already compiled Linux kernel using the new tools that aid memory forensic?

2.8 Contributions

We published the paper *Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots* and developed a framework to make Linux memory forensic viable for systems where the Linux kernel configuration is not known at analysis time. *Katana* generates the needed information for its analysis plugins from reliable patterns in the machine code of the Linux operating system under analysis. While this approach mispredicts around 20 percent of

data structure layouts, *Katana* correctly predicts the most important ones for forensic analysis. We validated this by evaluating 45 realistic memory snapshots taken from Android, various Linux distributions, and two IoT devices. The *Katana* framework succeeds in performing the classic forensic analysis tasks of *Volatility* on all snapshots without prior preparation, whereas *Volatility's* own analysis plugins fail due to imprecise debugging information. Examples of these classic analysis tasks are the *listing running processes*, *listing loaded kernel modules*, and *extracting the history of opened files*. *Katana* also offers an alternative implementation for grabbing the memory on the currently running system. The toolkit that was used by *Volatility* beforehand required knowledge of the Linux kernel configuration, which is not available in our assumed use case.

Furthermore, we published the paper *RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis*. This paper summarizes the current state of memory forensic frameworks including *Katana* and tested different anti-forensic techniques. The outcome is the *RandCompile* plugin for the GCC compiler. *RandCompile* adds further elements of *Software Diversity* into the Linux kernel to make an off-the-shelf kernel more resilient against common memory forensic frameworks. During our evaluation of *RandCompile*, we tested the limits of our *Katana* framework. *Katana* has shown to have a high resilience against simple manipulations of the ABI inside the Linux kernel. Additional strategies like adding bogus arguments to the ABI were required to defeat a considerable set of *Katana's* analysis methods. We conclude that anti-forensic is possible, but only with significant efforts for the operating system developer.

Last but not least, we suggested in *FridgeLock: Preventing Data Theft on Suspended Linux with Usuable Memory Encryption* a method to also protect the userspace memory from forensic actions if the machine is suspended. This tool serves two purposes. First it complements *RandCompile* which only aims at protecting data structures that are placed in kernel space. Second, it showcases that a security feature can be added to an already deployed Linux system using the Linux kernels own runtime information (*kallsyms* and *kprobes*, see Section 2.6), which are also used by *Katana* to enable the reconstruction of data layouts.

Chapter 3

Topic 2: Analyzing Real-World Threats through Honeypots

A honeypot is a dedicated machine whose sole purpose is to get attacked by an adversary. It is not used in normal operations and can, therefore, be safely compromised by an attacker. There are multiple reasons for a network operator to deploy a honeypot. First, he might deploy it in his network to distract an attacker from machines that contain actual production data. As honeypots should be attacked, they appear to an attacker less secured and are, therefore, usually attacked first. Since a honeypot is not used during normal operations, every incoming connection is part of an attack and can be easily classified as malicious. Therefore, the honeypot can serve as an early warning system for an ongoing attack [15].

Second, a network operator might want to analyse ongoing hacking attempts on his systems. He can deploy a honeypot and analyze the attempts of the attacker to break into the system. He can study the attacker's tools and methods, generate signatures for the malware deployed on the honeypot and feed them into the anti-virus system, or generate signatures of the network activity for the use in Intrusion Detection Systems.

It should be noted that using a dedicated machine enables the analyst to deploy additional monitoring, isolation, and logging mechanisms beforehand to even record attackers seeking to hide their traces. Because of this, they are a valuable tool for studying the threat landscape.

3.1 Honeypot Types and Their History

The first documented usage of a honeypot-like system dates back to 1986. Back then, Clifford Stoll, who administered a few computer systems at Lawrence Berkley Lab, discovered an attacker on one of the machines he was responsible for. Instead of deactivating the attacker's accounts on the production system, he decided to leave his account active and study his steps and methods to better protect the remaining computers on the network in the future [15]. It should be noted that the system the attacker gained access to was a production system and, therefore, according to the above definition, not a honeypot. However, the idea of a system whose sole purpose is to study attackers and their methods was born.

In 1999, the Honeynet Project was founded as a non-profit security research organization which aimed to support research in the area of honeypots, raise awareness, and develop honeypot

tools to gain insights on real-world attacks in the public internet [16]. The Honeynet Project is still active today and has published many honeypot implementations and threat analysis tools over the years, including *Honeyd*, *Glutton*, and *Sebek2*.

During the long development period of honeypots, different types have evolved, which we will describe in the following:

Low-Interaction honeypots are honeypots that are simple and easy to deploy. Usually these honeypots are a reimplementation of software that is known to be subject of attacks. A known vulnerability might be emulated in the reimplementation so that it can be exploited by an attacker. Furthermore, additional logging functionality is commonly added to the honeypot, which simplifies the post-attack analysis. However, the reimplementation is rarely feature complete. Therefore, an attacker can detect the different behaviour of the low-interaction honeypot and abort his attack.

High-Interaction honeypots aim to mimic a lifelike system as closely as possible. They often modify genuine system software to contain additional logging and monitoring capabilities or are implemented as proxies, making minimal changes to the protocol exchange. Therefore, the honeypot should not be distinguishable from the original software. However, as high-interaction honeypots are based on the full original software, they are harder to build, need more resources than their low-interaction counterparts and are also harder to maintain. High-Interaction honeypots might utilize virtualised machines with virtual machine introspection (see Section 4.1) to isolate the logging and monitoring system from the vulnerable software stack.

Related to classical honeypots are *Tarpits*. *Tarpits* are honeypot-like software packages that are designed to slow down an adversary. This can already be done at the networking level by delaying acknowledgement messages of incoming network packets. However, *Tarpits* can also be implemented at the application level. This has been done for the HTTP and SMTP protocol, to delay harvesting e-mail addresses and the delivery of spam through SMTP servers.

A significant amount of research has already been performed in the field of honeypots. They have been used at client- [17] and server-side [15], [18]–[20] to analyze threats. For example, the client-side honeypot created by Wang *et al.* [17] crawled the internet for webpages that exploit the user's web browser. An example of a server-side honeypot is the *Dioneaea* honeypot, which emulates a number of network protocols like MySQL, Microsoft's SMB, and FTP. The emulated version of Microsoft's SMB protocol is still vulnerable to CVE-2017-0143 (known as the WannaCry vulnerability) and respective malware uploaded to the system is saved for later analysis. Likewise, received malware for the other protocols is captured.

Honeypots have also been adapted to capture threats to Industrial Control Systems and IoT devices [21], [22]. Furthermore, they are also deployed in industry. Akamai, a commercial provider for cloud and content delivery services, offers an industry product that involves honeypot technology [23].

3.2 Challenges

Running and deploying a honeypot comes with its own set of challenges. First, attackers come and go and their tools change. A honeypot deployed today may attract different attackers with different goals than it did years ago. Second, depending on the number of attacks, a honeypot may need to be deployed for a period of several months to collect statistically relevant results.

Third, because honeypot systems have intentional weaknesses, special care must be taken to properly sandbox the honeypots so that they do not pose a threat to other systems on the same network or to other systems on the Internet. In some cases, these risks may be legal rather than technical. Consider a threat actor such as North Korea, a country subject to international sanctions and embargoes. Is a company funding such a state by allowing it to mine cryptocurrency on a honeypot system? While a good access control system can limit the resources an attacker can access on the honeypot, honeypots are typically not monitored 24 hours a day, 7 days a week. There remains a risk of attackers using a previously unobserved attack vector to circumvent the restrictions and abuse the honeypot for some time. During our experiments, we interviewed representatives of a DAX company who were very hesitant to take the risk due to liability issues.

Forth, because high-interaction honeypots are often patched from real software rather than rewritten from scratch, maintenance requirements increase, as does the overall complexity of the system. This is especially true when the monitored software changes regularly due to updates, requiring close human supervision during operation.

Despite the amount of research that has been done on honeypots, many of the open source honeypot implementations available on GitHub are outdated and no longer maintained. In an experiment conducted in the early days of this research, *Dioneaea* collected malware samples that were *corrupted* and *not* executable. While it remained unclear if the malware was already sent out corrupted by the attacker or was corrupted during the collection, the very limited amount of up-to-date studies on the prevalence of honeypots in the Internet, documentation, and resources on this topic suggested further research. While there is a single study by Vetterl *et al.* [24] which provides measurements on the prevalence of honeypots for the FTP, HTTP, and SSH protocols, there is no study available for widely used Microsoft-based protocols like SMB and RDP.

This leads us to our next research question:

> **Research Question 3**
> What honeypots are currently used in the public internet, how do they operate, and how common are they?

Furthermore, we observed that there is a lack of modern studies on how cautiously attackers act to avoid honeypot deployments. Because of the high maintenance effort, honeypot operators want to avoid maintaining a high-interaction honeypot where possible.

3.3 Contributions

In the paper *Looking for Honey Once Again: Detecting RDP and SMB Honeypots in the Internet* we answer the third research question. We conducted an internet-wide scan to detect known and unknown honeypots for two major protocols of the Microsoft Windows operating system. The RDP protocol is used by Microsoft Windows to offer remote desktop services and SMB is commonly used to exchange files between computer systems in a cooperate network. We utilized differential fuzzing to find a set of distinctive packets that can detect the known honeypots. Furthermore, we probed all available versions of the Windows operating system. Afterwards, we used this information to scan our collected dataset for abnormal answers which could indicate the presence of a honeypot.

In our experiments we could clearly identify several known honeypots in the internet. Further, we identified those systems that do not use Microsoft's implementation *Schannel* for building up RDP TLS connections, but the *OpenSSL* library. While the latter is very popular in the open-source community, this library has never been used by Microsoft in a any implementation of RDP. Our verification showed a high reliability (99 percent correct) of our scanning classifications.

Further, we set up multiple honeypots ourselves for the RDP, HTTP, and SSH protocol to check if attackers change their behaviour depending on the honeypot used. While we have not been able to fully validate our results due to insufficient data, our results suggest that stealthy high interaction honeypots receive more sophisticated commands than low-interaction honeypots. Likewise, the amount of packets exchanged with the attackers is higher than on low-interaction honeypots. Human attackers are more likely to abort attacks on low-interaction honeypots than automated tooling. While this was to be expected, we reconfirmed this hypothesis on today's threat landscape. We observed the following attacker types:

1. **Credential Stuffing Attacks:** Some attacker just tested various credentials and immediately disconnected upon a successful login. These attack seem to happen regardless of the honeypot technology used.

2. **Querying OS and Machine Details:** Some attackers logged into our system, checked the operating system / kernel version installed, and collected information about the CPU and memory resources available on the machine. These attacks seem to happen regardless of the honeypot technology used (even if an attacker was able to detect the honeypot in the early stages of the protocol).

3. **Crypto Concurrency Mining:** Other attackers installed mining software to utilize the machine resources to mine crypto concurrencies like Bitcoin. This is less likely on low-interaction honeypots.

4. **Botnet/Proxy For Other Attacks:** Some attackers tried to add the respective honeypots to their botnet or used the hosts as a proxy to attack other servers / websites. This is less likely on low-interaction honeypots.

Chapter 4

Research Methods

In this chapter, a fundamental overview about the relevant research methods for this thesis will be given. For the publications *Katana: Robust, Binary-Ony Forensic Analysis of Linux Memory Snapshots* and *RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis* the most relevant concepts are described in Section 4.1, Section 4.2, and Section 4.4. For the construction of new honeypots and for the comparison to existing deployments in the Internet, the core methods are outlined in Section 4.5 and Section 4.6.

4.1 Virtual Machine Introspection

Virtual Memory Introspection (VMI) was originally introduced as an intrusion detection mechanism by Garfinkel *et al.* [25] in 2003. The problem they faced was that an intrusion detection software system running on the same host that the system is monitoring could be compromised at the same time as the host and therefore not be able to raise an alarm about the attack. On the other hand, a network based intrusion detection system is less effective and, therefore, also not an optimal solution, because it can only monitor the traffic and not the interactions of potential malware with the host.

Their solution was to place their malware detection system in the Virtual Machine Monitor (VMM) on the host. In order to enforce a separation between virtual machines the VMM is isolated from introspection and manipulation attempts of the guest VMs running. However, in contrast to a network based solution, it is able to inspect not only communication of the VM that is leaving the host, but also inspect and manipulate the CPU registers and main memory. According to Garfinkel *et al.* [25], the structure of the operating system is known *a priori*. Therefore, the VMI system is by definition aware of the memory layout for interacting with the operating system data structures.

Notable frameworks for VMI are *LibVMI* [26] and its predecessor *XenAccess* [27], [28]. Furthermore, there is the toolkit *Panda* [29], which offers a framework for building dynamic analysis for virtualised systems (see Section 4.2). The existing plugins for operating system introspection of the *Panda* toolkit can be expanded with own plugins.

Besides these toolkits for building up own analysis systems, there are also fully developed VMI systems like the *Drakvuf Sandbox* sandbox for stealthy malware analysis [30], [31]. Furthermore, there is the *WhiteRabbit* VMI framework [32], which injects itself into an already deployed

system. Furthermore, while not dedicated to the inspection of virtual machines, but to systems of all kind, the *Volatility* and *Rekall Forensic* toolkits are also closely related to the field of VMI.

4.2 Automated Program Analysis

A program analysis is a formal and algorithmic reasoning process over a given computer program. In general, this kind of analysis is necessary, when one program needs information about the *behaviour* of another one. A typical example most computer scientists will be familiar with is a compiler. A compiler reasons about the source code and in case of an optimizing compiler transforms it into machine code. As a programmer might have left parts in the program that are not in active use and, therefore, a optimizing compiler will performs a dead code analysis over the different functions to eliminate dead code.

Besides compilation, program analysis has also many other applications such as correctness proofs, reverse engineering, vulnerability scanning, and other automated tests for bugs. There are plenty notable scientific examples of static analysis tools. In 2016, Stephens *et al.* [33] proposed *Driller*, which leverages fuzzing and selective concolic execution to find and exploit memory corruption bugs in the DARPA Cyber Grand Challenge. Furthermore, the *LLVM Project* and the *GNU Compiler Collection Project* both released static analysis tools to find bugs in C and C++ programs.

Program analysis can be conducted using static or dynamic methods. However, not both methods are suitable for every use case.

Static Analysis works by just analyzing a given program *without running it*. Therefore, all possible cases are considered in the analysis. While this method works well for small problem instances, it quickly becomes computationally infeasible if the analysis should be conducted on a complete program or a set of communicating programs. In practice, this leads to a over approximation of possible program paths and values and, oftentimes, therefore to false positives. Furthermore, methods such as symbolic execution suffer from the path explosion problem.

Dynamic Analysis works by analyzing a program *currently running*. Usually, the program is instrumented (i.e. the original source code or machine code is modified at predeemed points of interest) and than executed. In contrast to static analysis tools, only inputs that can also occur in a given part of a program. This usually reduces false negatives in program analysis while discarding cases that occurred in runs that have not been observed.

Usually program analysis includes multiple analysis steps for the purpose of a bigger reasoning. Classical steps are:

Control-Flow Analysis aims to transform a given program into a control-flow graph where the nodes are locations in the program. All edges in these graph connect other nodes which

can be reached from this specific location in the program. Instructions inside the program can be divided into control flow changing and non control flow changing instructions. Non control flow changing instructions are defined as instructions that only have a single succeeding instruction *regardless* of the current execution context. To simplify the graph, a series of non control flow changing instructions form a *basic block* that shape the nodes in the control flow graph.

Data-Flow Analysis aims to transform a given program into a data-dependence graph. In this graph, nodes are usually formed by the variables occurring in the program. An edge depicts the flow of information from a source variable or result of a function into another variable. A flow of information is present if the value of a variable can influence the value of another variable. A data-flow analysis requires a prior control-flow analysis [34].

Code-Coverage Analysis is performed in dynamic program analysis settings. It tracks what execution paths in the control flow graph have been *covered* by the dynamic analysis. Dynamic methods, such as program testing through *Fuzzing* usually seek to increase coverage to increase the completeness of the analysis.

4.3 Binary Lifting

To ease program analysis for compiled binary programs to a set of different CPU architectures, the machine code is usually lifted to a *intermediate representation*. In turn, this intermediate representation is than analyzed. Therefore, all program analysis algorithm only need to support the single *intermedia representation* and do not need to deal with the complexity of each instruction set. Because of this, this concept is frequently used by decompilers that aim to transform the compiled machine code representation of various different instruction sets back to semantically equivalent source code. As the source code form is usually easier to read and understand, decompilers are valuable tools for reverse engineers.

We based all of *Katanas* program analysis functionality on top of the open sourced Ghidra decompiler, which uses P-Code as intermediate representation. P-Code has support for many popular CPU architectures out there, such as X86, x86-64, ARMv7, ARM64, MIPS, Sparc, PowerPc and many more. However, this is not the sole existing *intermediate representation*. Over the years, multiple evolved based on the needs of the performed analysis tools build upon. Other notable *intermediate representations* are LLVM-IR, which is used internally by the LLVM Compiler Project, and VEX, which is used by the *angr* program analysis framework.

It should be noted, that lifted code can also be used to *transpile* (i.e. swap the instruction set of a program) from one instruction set to another one.

4.4 Software Diversity

The concept of Software Diversity has already been suggested by Forrest *et al.* [35] back in 1997. Software diversity targets to reduce the potential attack vectors that arise from the fact that software functions in the exact same way on all deployed systems. Small variations might be sufficient to render an exploit for a memory corruption vulnerability unusable. In the Linux kernel KASLR (see section 2.3) is a widely deployed and Structure Layout Randomization an optionally available defense (see section 2.4).

Software Diversity can be applied at compilation time or at runtime. Usually solution operating at runtime will increase the startup time of the respective program in order to perform the randomization. However, compilation time solutions will randomize the layout only once and not on every startup. As Software Diversity relies on the secrecy of the concrete randomization applied, the diversified binary cannot be made distributed to multiple users or made public. It is necessary, that every user creates their on individual binary of the software and that an attacker cannot easily access the binary on disk before the attack. While this might be time consuming, it is still feasible on server and desktop machines, but infeasible on resource constrained devices (such as IoT devices or smartphones) [36]. We would like to remind the reader, that these is one of the reasons while structure layout randomization is not commonly deployed: It would require every user to compile his own Linux kernel, which would require a substantial amount of CPU resources. A potential solution to this problem a hybrid approaches, which generate metadata at compile time that enable a rapid diversification at installation time [37].

In the past decades, also other ways to create Software Diversity in operating systems and user space programs have been discussed:

Introduction of NOP instructions The insertion of nonfunctional code by the compiler will result in a different binary layout of the executable code. This makes ROP attacks harder as the attacker cannot guess the exact position of the ROP gadgets in the binary without an additional information leakage that reveals pointers to the executable code [35].

Reordering of code parts In many cases a compiler can reorder the basic blocks at control flow branches. Instead of basic blocks, it is also possible to reorder the linkage of whole functions at runtime. This has been proposed and implemented for the Linux kernel [38] in 2020, but is not a mainlined feature as of today.

Reordering memory layout This approach is taken by Structure Layout Randomization. But instead of the data layout of structures, also other parts of the memory layout can be changed. Forrest *et al.* [35] suggests to add gaps on the stack of the program. This would influence the relative offsets of the local variables stored on it. As further addition, they suggest to transform a program so that the stack frames get randomly allocated on the heap.

Hiding values by XOR encryption Pointer values can be encrypted with a cheap encryption operation (like XORing it with a random key) to hide them. This has been done by Williams-King *et al.* [39] to randomize the return addresses of a patched operating system. In his approach these addresses would have been artifacts that might would have been useful to an attacker.

It should be noted that the line between Software Diversity and Software Obfuscation is blurry. Larsen *et al.* [36] name also classic obfuscation techniques as Control Flow Flattening as method to archive Software Diversity.

The concepts of software diversity are heavily used in our research paper *RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis*, where we discuss them as a potential defense mechanism against forensic tools. In order to operate, these tools make a lot of assumptions about the inner workings of the binary and, in case of *Volatility* and *Rekall Forensics* heavily rely on precomputed information from static distributed binaries.

4.5 Internet-wide Scans

To study real-world software deployments of internet-connected software, a systematic scan of the entire IPv4 address space is a viable option to gather high-quality research data. While the IPv4 address space could contain up to 2^{32} hosts, a systematic scan of the whole address space has become feasible on a machine with a high-speed gigabit internet connection [40]. Furthermore, the number of routable and usable IPv4 addresses in the public internet is lower than 2^{32}, which further reduces the effort needed for scanning. E.g. not all subnets advertise a route and are, therefore, not reachable from the vantage point from where the scan is conducted. Other IPs are reserved for special purposes (e.g. broadcast addresses) and cannot be used to address a host.

There are several examples of previous internet-wide scans that seek to measure security across the internet. First, there is the *EFF SSL Observatory* [41], which contains a dump of all publicly used HTTPS certificates in 2010. Second, the authors of the *LogJam* vulnerability performed an internet-wide scan to evaluate how frequent weak Diffie-Hellman parameters are on HTTPS and SSH servers on the public internet. This research influenced the security settings of several major internet browsers [42].

As internet-wide studies have shown to be powerful, several online-services have evolved that conduct internet-wide scans and publish their results. Such services are, for example, *Censys* and *Shodan.io*. Unfortunately, these services do not collect all data we needed for our exploratory study about honeypot deployments in the internet. Where applicable, we compare their data to our own data for verification, but have collected our own data using the *ZMap* scanner [40]. *ZMap* performs a TCP-SYN scan of selected TCP ports on the target hosts. It generates the list of IPs to scan via a generator function that enumerates the overall IPv4 address space in

a non-sequential order. This avoids traffic peaks in the network of single organisations that might have the capacity to deal with a fast internet scan. Responsive IPs with an open port are reported to a second scanning stage that performs a deeper analysis. This second stage consists in the research paper *Looking for Honey Once Again: Detecting RDP and SMB Honeypots in the Internet* out of a custom Python scanner that sends a set of distinctive packets to classify the host. The answers and classifications are saved for later analysis.

4.6 Differential Fuzzing

Fuzzing has become a popular software testing technique especially in the security domain and gained a lot of interest from industry and research. Google started the OSS-Fuzz project back in 2016, which is continuously fuzzing more than 1000 software projects since then [43]. The concept of fuzzing is straightforward: a fuzzer generates a random input, feeds this input into the program to test, and memorizes inputs that have resulted in a crash or other abnormal behavior. In its simplest form, fuzzing is straightforward to implement and use, yet it remains surprisingly effective, which is probably the reason why it has become so popular. Nevertheless, a significant amount of time and resources have been dedicated to the development of more sophisticated techniques for generating inputs that are more likely to result in a crash, to enhance the rate at which new inputs can be tested, and to group related crashes.

In contrast to usual fuzzing, differential fuzzing feeds the same input into multiple different program implementations. Input that causes different behaviour in different implementations indicates an error in one implementation [44]. This strategy for finding and pining down errors in computer programs is less frequently used. However, one use case has been in the past the testing of C compilers [44], [45]. Likewise as in conventional fuzzing, the effectiveness of this testing methods depends on the quality of the chosen inputs.

In the field of security research, *Differential Fuzzing* has proven to be useful for side-channel analysis (see *DifFuzz* [46] and *Stacco* [47]). Side-Channels must be avoided in cryptographic algorithms as small differences in memory consumption or timing may result in loss of secret key material and, therefore, to a catastrophic failure of security.

We have adapted the concept of *Differential Fuzzing* for the use in our research paper *Looking for Honey Once Again: Detecting RDP and SMB Honeypots in the Internet* to identify differences between the respective honeypot implementations and the original implementation. We tested two categories for fuzzing. At first, a protocol aware fuzzer that knows about the layout of the network packets and can choose meaningful values for the different fields in each packet. Second, we developed a fuzzer that records and replays parts of the network communication. During the replay, random bits are applied to the sent network packet. Both fuzzers send their crafted packets to the benign implementation and the other to the respective honeypots. A varying answer from one of the parties denotes a payload that could qualify as a *most distinctive packet*, which is a packet designed to create the maximum number of different answers between

all benign and honeypot implementations. A similar approach for the detection of honeypots has been taken by Vetterl *et al.* [24], however we have extended the methods to the RDP and SMB protocols, which are more complex. For more details about this research, see Chapter 7.

4.7 Binary Instrumentation

Binary instrumentation describes a technique to modify an existing program by inserting additional code for additional functionality into it. This code can modify the programs original behavior, analyze or monitor it. Like program analysis, binary instrumentation can be done in two main ways: statically and dynamically. Static instrumentation changes the binary file on disk, while dynamic instrumentation modifies it while the program is running. Binary instrumentation can alter the behaviour of programs that haven't been prepared for it.

Instead of instrumenting the binary executable, instrumentation can also be performed at compilation time. In this case, the compiler inserts patch points, where a dynamic system can later divert control flow or apply a needed instrumentation directly. Adding the transformations at the compiler stage, has semantic and performance advantages, which has proven to be useful for fuzzing and profiling applications. Unfortunately, in practice source code is not always available so compile time instrumentation is not always an option [48].

Examples for *dynamic* binary instrumentation (DBI) frameworks for userspace programs are Intel PIN, Frida, DynamicRIO, or Valgrind. These tools have been developed with different use cases in mind. Frida targets the community of reverse engineers and security researchers [49], while Valgrind has been designed as a heavyweight instrumentation framework with support for *shadow values*. These *shadow values* enable Valgrind, for example, to search for undefined variables at runtime [50]. Depending on the tool used, there is also support for attaching to a process already running (i.e. this is supported by Frida and Intel PIN).

This is particularly useful for creating a stealthy honeypot of an existing windows service (like the remote desktop service). For our research on honeypots, we used a DBI framework to monitor the existing RDP service, which handles remote desktop connections, of Windows. We added additional probes to record the screen that is rendered out to attackers and to record their keystrokes and mouse events. As the original implementation of the service is the original implementation of Microsoft, we deem this honeypot to be indistinguishable from a regular Windows RDP service.

DBI was also used for our work on *FridgeLock*. The kernel development community has created their own dynamic instrumentation mechanisms, because the DBI frameworks being presented so far are unusable inside an operating system. The kprobes feature of the kernel allows code to be added to the beginning of most Linux kernel functions (except functions declared as private within a compilation unit, which can be inlined into other functions by the compiler). Complementary, *kretprobes* can be used to add code that will be executed at function exit.

By using these two features, we can realize many features without explicit support of Linux (see Chapter 7 for details).

4.8 Microbenchmarks

For a wide acceptance of security hardening mechanisms, its performance impact plays a crucial role. Users often prioritize systems that not only offer robust protection but also operate seamlessly without significant performance degradation. Therefore, we evaluated also the performance overhead of security defenses that evolved during the course of this thesis. In the proof of concept implementation in our research paper *RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis* we use the *lmbench* microbenchmark [51].

A microbenchmark, such as lmbench, focuses on measuring the performance of specific components or subsystems within a larger system. Unlike macrobenchmarks that evaluate overall system performance, microbenchmarks delve into the intricate details of individual functionalities. In the case of lmbench, it meticulously scrutinizes various aspects of the Linux kernel's performance, offering insights into CPU, memory, and I/O operations. It benchmarks latency and throughput of various system calls.

Chapter 5

Core Publications

This chapter contains a one-page summary of each core publication in chronological order.

5.1 FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption (ACM CODASPY 20)

Abstract

To secure mobile devices, such as laptops and smartphones, against unauthorized physical data access, employing Full Disk Encryption (FDE) is a popular defense. This technique is effective if the device is always shut down when unattended. However, devices are often suspended instead of switched off. This leaves confidential data such as the FDE key, passphrases and user data in RAM which may be read out using cold boot, JTAG or DMA attacks. These attacks can be mitigated by encrypting the main memory during suspend. While this approach seems promising, it is not implemented on Windows or Linux. We present FridgeLock to add memory encryption on suspend to Linux. Our implementation as a Linux Kernel Module (LKM) does not require an admin to recompile the kernel. Using Dynamic Kernel Module Support (DKMS) allows for easy and fast deployment on existing Linux systems, where the distribution provides a prepackaged kernel and kernel updates. We tested our module on a range of 4.19 to 5.3 kernels and experienced a low performance impact, sustaining the system's usability.

Contributions of the Author

The author invented the idea and concept of a Linux kernel module that hooks into the suspend API of Linux and performed the encryption of the RAM there. While the idea of suspend time memory encryption was not new, we were the first who added this feature to the Linux kernel without the need of recompiling the whole Linux kernel. Furthermore, he was heavily involved in evaluating the final implementation, the paper writing and editing process.

`https://dl.acm.org/doi/10.1145/3374664.3375747`

5.2 Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots (RAID 22)

Abstract

The development and research of tools for forensically analyzing Linux memory snapshots have stalled in recent years as they cannot deal with the high degree of configurability and fail to handle security advances like structure layout randomization. Existing tools such as Volatility and Rekall require a pre-generated profile of the operating system, which is not always available, and can be invalidated by the smallest source code or configuration changes in the kernel. In this paper, we create a reference model of the control and data flow of selected representative Linux kernels. Using this model, ABI properties, and Linux's own runtime information, we apply a configuration- and instruction-set-agnostic structural matching between the reference model and the loaded kernel to obtain enough information to drive all practically relevant forensic analyses. We implemented our approach in Katana, and evaluated it against Volatility. Katana is superior where no perfect profile in- formation is available. Furthermore, we show correct functionality on an extensive set of 85 kernels with different configurations and 45 realistic snapshots taken while executing popular Linux distributions or recent versions of Android from version 8.1 to 11. Our approach translates to other CPU architectures in the Internet-of-Things (IoT) device domain such as MIPS and ARM64 as we show by analyzing a TP-Link router and a smart camera. We also successfully generalize to modified Linux kernels such as Android.

Contributions of the Author

Idea, Design, concept, and first proof of concept implementations of *Katana* have been created by the author. Later on, the co-author Manuel Andreas suggested and implemented a P-Code extension for the structural matching. Co-Author Tobias Holl implemented the Module-based Snapshot Creation mechanism, Julian Kirsch helped out in the early reversing parts needed to implement the match algorithm. Furthermore, the author was heavily involved in writing the final text for the paper, editing, and designing and performing the final evaluation.

Copyright & Link to Original Publication

`https://dl.acm.org/doi/abs/10.1145/3545948.3545980`

5.3 Looking for Honey Once Again: Detecting RDP and SMB Honeypots on the Internet (Euro S&PW 22)

Abstract

Honeypots are a widely used technique to observe the spread of malware and the emergence of new exploits. Attackers are said to avoid connecting to honeypots as they reveal the attacker's methods, tools, and exploits. However, there is limited proof for that assumptions in the literature. In this study, we will fingerprint existing honeypot implementations and develop new ones that are not detectable by the fingerprints generated. While different honeypot implementations have been fingerprinted in the past, we see a lack of studies covering Windows-related protocols such as Remote Desktop Protocol (RDP) and Server Message Block (SMB) honeypots. However, these protocols have seen at least two major security vulnerabilities in the past 5 years and are commonly exploited. We adapted existing fingerprinting algorithms to allow an accurate identification of RDP and SMB honeypots checking how implementations behave in error conditions. We present a new improvement, namely the inclusion of system TLS stack features previously not used for honeypot detection. We are the first to perform an internet-wide scan searching for RDP and SMB honeypots. We are able to effectively uncover the presence of two common open-source honeypots for RDP and SMB each. We identified 84 instances of Heralding (RDP), 1123 instances of RDPY (RDP), 60 instances of Impacket (SMB), and 1461 instances of Dionaea (SMB) during our scans. Furthermore, we found several hosts, which do not use Microsoft's SChannel TLS stack, but advertise themselves as Windows machines. This indicates the presence of a Man-in-the-Middle (MitM) box and could be a sign of a honeypot. Eventually, we analyzed how attackers interact with detectable honeypots. We deployed instances of RDP honeypots ourselves and found that credential guessing attackers seem to avoid them. This proves that RDP and SMB honeypots are fingerprintable and that even MitM-box-based high-interaction honeypots leave detectable traces.

Contributions of the Author

The author developed the differential fuzzer for the RDP protocol used in this study. Furthermore, he obtained the fingerprints of all RDP honeypots and derived a set of distinctive packets to uncover honeypots. He was heavily involved in the editing process of the paper and performed the validation part.

Copyright & Link to Original Publication

https://doi.org/10.1109/EuroSPW55150.2022.00033

5.4 RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis (ACSAC 23)

Abstract

Recently proposed tools such as LogicMem, Katana, and AutoProfile enable a fine-grained inspection of the operating system's memory. They provide insights that were previously only available for Linux machines specifically instrumented for cooperation with virtual machine introspection frameworks. An overly controlling cloud operator can now regularly deep-inspect VMs under their control. In this paper, we investigate how the concept of software diversity can be employed to remove structural information from the Linux kernel to harden it against automated analysis by the aforementioned tools. Furthermore, we give a systematic overview about the methods used by common automated memory forensic frameworks to gather their data. We found that all of them rely on a small set of features to provide their functionality. We employ a mixture of small targeted obfuscations to the memory layout and randomization of the ABI between functions in the Linux kernel as they provide predictable artifacts across different compilers, kernel configurations and the presence of Structure Layout Randomization. We provide an implementation of our ideas in RandCompile, which is composed of a small patch set for the 5.15 Linux LTS kernel and a compiler plugin. RandCompile seeks to remove structural information artifacts, which we call forensic gadgets, to eliminate all leverage points for further analysis of the tools mentioned above. Our approach does not require major modifications to the kernel code base and only has a negligible performance impact (less than 5% percent), which is less than other major security or debugging features enabled by default in the Linux kernel.

Contributions of the Author

The author had the initial idea and designed the implementation concept of RandCompile based upon a GCC plugin. Co-Author Andreas Chris Wilhelmer implemented the parameter order randomization, which has been completed by the author with the addition of pointer encryption and printk format externalization. Furthermore, the author conducted the literature survey and systematization of knowledge of other forensic frameworks and performed the final evaluation of RandCompile's performance against their analysis methods. He was heavily involved in creating and editing of the final paper submitted to the conference.

Copyright & Link to Original Publication

`https://dl.acm.org/doi/10.1145/3627106.3627197`

Chapter 6

Final Remarks

6.1 Recent Developments

Since the development of the *Katana* framework, other works of interest have been published in the area of *Memory Forensics*. In 2023, the *Fossil* tool [52] was proposed. The authors suggest an OS-agnostic way to memory forensics. They observed that core data structures like linked lists are implemented in consistent patterns with only minor deviations across different operating systems. By scanning inside the memory for these patterns, they have been able to reconstruct the process list of Windows, Linux, macOS and other lesser known operating systems using the same algorithms. While the usage of *Fossil* does not require handwritten rules (as *Katana* does), it still requires that the analyst knows a seed value. This is a value that is definitely present in the container data structure the analyst looks for. In case of the process list, this could be the name of a single process. This is a notable improvement that will ease the analysis, but so far *Fossil* is not capable of performing analysis that requires deep knowledge of the data structures. An example of such an analysis is the listing of recently opened files, which *Katana* can perform.

In his PhD thesis, Oliveri [53] pointed out that the developed platform independent memory acquisition module of *Katana*, which we have created as successor to *Lime*, does require the disabling of the interrupts of the system. This could stop the system from correct operations in some cases. This is true, but rarely a limitation in practice. A system which is suspected as compromised rarely needs to function normally, but is disconnected from the operational business network beforehand.

In 2023, Spahn *et al.* [19] conducted an internet-wide scan looking for vulnerable and publicly available instances of container orchestration tools. They found vulnerable instances of *dockerd*, Kubernetes and Apache AirFlow instances. In a second step, they designed a high-interaction honeypot to capture attackers themselves. Their honeypot setup for Docker and Kubernetes involved patching the Docker source code to insert additional logging and tracing functionality. The Apache AirFlow honeypot was realized through a transparent TLS proxy that re-encrypts all traffic. They recorded a considerable amount of scanning traffic and attacks that aim to mine cryptocurrencies.

6.2 Conclusion

In the following section we will recap the research questions of Chapter 2 and Chapter 3 and discuss our results.

> **Research Question 1**
> Can memory forensic tools be built which do not need to know the exact Linux kernel configuration and which would even work in the presence of Structure Layout Randomization (or other defense mechanisms based on *Software Diversity*)?

We proofed this by building *Katana*. While the reconstruction of data structure layouts performed by *Katana* is less precise than old approaches based on debugging information, it is better than all solutions available if a kernel configuration is *not* available for forensic analysis. We verified this hypothesis on a set of 85 kernels with differing configurations. If the Linux kernel configuration does not match the *Volatility* profile *exactly*, *Katana* will achieve better analysis results. Additionally, we tested 45 memory snapshots of popular Linux distributions and showed that *Katana* can perform all classical analysis tasks of *Volatility*.

The recent developments in the area of Memory Forensics, outlined in Section 6.1, further stress the academic interest in these kinds of tools. While the approach presented by Oliveri *et al.* [52] is promising, they are not aware of the specifics of the Linux kernel and can, therefore, not perform the same kind of in-depth structure analysis that *Katana* is capable of.

We also found that forensic research could have applications in reverse engineering since *Structure Layout Randomization* is sometimes enabled by vendors to obfuscate their modified Linux kernels. Aside from that, forensic has the potential to release reverse engineers from a legal burden. According to German and European law, analyzing software by observation is legal in all cases [54]. Consider the case of a Linux-based IoT device with a proprietary user space application. While legally obliged, the manufacturer might be reluctant to provide the kernel sources or configuration used. A forensic tool could modify the Linux operating system of the IoT device and directly observe the running process with forensic methods or inject further monitoring functionality to allow for an in-depth analysis of a proprietary user space application. As the Linux kernel is licensed under the General Public Licence (GPL), this testing setup should not violate any copyright claims of the device vendor.

> **Research Question 2**
> How easily can we add a security feature to a deployed (preferably without knowing its configuration), already compiled Linux kernel using the new tools that aid memory forensic?

While developing *Katana*, we created a memory acquisition module that can be built and injected into the kernel without the need for the exact build environment. This is a novel feature that is not provided by classical Linux kernel modules. Further, we developed *FridgeLock* to add suspend time memory encryption to an already compiled Linux kernel. In contrast to the memory acquisition kernel module, *FrideLock* showcases the addition of a more complex security feature into an already deployed system. It can dynamically adapt to the running host kernel (e.g., if the kernel retrieves a security patch) using the Dynamic Kernel Module Support (DKMS) framework of Linux. We conclude that it is indeed possible to add non-trivial security features to an already compiled and deployed Linux system. This technique can be used in environments where the system administrator does not want to maintain his own fork of the Linux kernel, which, for example, necessitates a build server and an own package repository. However, the maintenance effort of a separate dynamic kernel module is likely too high to be practical in many application scenarios. The kernel would benefit from a more comprehensive and stable API to its internal core components. For example, when we designed *FridgeLock*, the kernel did not provide an API to access the currently loaded disk encryption keys, which is essential to perform user space encryption before the system is suspended. Furthermore, the Linux kernel still does not allow dynamically loaded Linux Security Modules (LSMs). LSMs offer a way to implement fine-grained access control policies on a Linux system as the LSM is queried during almost all file operations. While existing LSMs like *SELinux* [55] and *AppArmor* [56] are quite comprehensive, we found it difficult to modify their standard policies in practice. In many scenarios, it is easier to program the rules instead of writing them in the domain-specific language of existing LSMs.

With the advent of eBPF [57], the situation improves. With the release of Linux 5.7, a LSM based upon eBPF exists that allows dynamic additions of security policies to the Linux kernel. This feature has already been adopted by companies like CloudFlare in production in favor of classic LSMs [58]. Eventually, our implementation strategy still fills a gap where security features are urgently needed and a matching Linux API does not exist yet.

> **Research Question 3**
> What honeypots are currently used in the public internet, how do they operate, and how common are they?

We conducted an internet-wide scan to detect known and unknown honeypots for two major protocols of the Microsoft Windows operating system during our research for the paper *Looking for Honey Once Again: Detecting RDP and SMB Honeypots in the Internet*. We were able to identify around 2600 active honeypot instances that different parties deployed. While several institutions like Deutsche Telekom publicly announce that they operate honeypots [59], the exact number of similar honeypot installations was previously unknown. There might be further instances placed in firewalled subnets to which we could not connect during our scan.

Additionally, we found that the majority of honeypots are deployed within the network of cloud providers. The cloud provider networks containing the highest prevalence of honeypots are Amazon AWS, followed by Choppa.com, DigitalOcean, and NetCup (a German cloud hosting provider known for lax firewall filtering policies).

We further found multiple signs of unknown high-interaction honeypots for the RDP protocol. On a cluster of machines, the RDP protocol is implemented using an *OpenSSL* stack to perform the TLS encryption while the rest of the protocol follow the Microsoft specification of the protocol precisely. This is abnormal, because Microsoft uses its own TLS implementation *Schannel* for RDP, which behaves different than *OpenSSL*. We conclude that the operator of these machines installed a TLS interceptor based on *OpenSSL* that forwards the traffic to the respective RDP servers. Please note that this requires the TLS interceptor to exchange the host certificates of the server in upper protocol layers as well and is not trivial. We could not verify if the network operators set up the intercepting machines as honeypots or for other purposes. The recent developments show that other research groups also try to understand the threat landscape by using very similar methods. This yields the conclusion that high-interaction honeypots are build by modifying benign software to limit detection possibilities.

However, there is still room for further research: With the help of program analysis, the methods of machine learning, and AI, a given program might automatically be converted into a high-interaction honeypot. This would reduce the significant initial effort of creating honeypots and allow for faster development.

Chapter 7

Published Version of Papers

In the following chapter, the author's versions of all core publications of this thesis, as they have been submitted to the respective conference venues, are reprinted in chronological order.

KATANA: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots

Fabian Franzen
Technical University of Munich
Munich, Germany
franzen@sec.in.tum.de

Tobias Holl
Technical University of Munich
Munich, Germany
tobias.holl@tum.de

Manuel Andreas
Technical University of Munich
Munich, Germany
manuel.andreas@tum.de

Julian Kirsch
Technical University of Munich
Munich, Germany
kirschju@sec.in.tum.de

Jens Grossklags
Technical University of Munich
Munich, Germany
jens.grossklags@in.tum.de

ABSTRACT

The development and research of tools for forensically analyzing Linux memory snapshots have stalled in recent years as they cannot deal with the high degree of configurability and fail to handle security advances like *structure layout randomization*. Existing tools such as *Volatility* and *Rekall* require a pre-generated profile of the operating system, which is not always available, and can be invalidated by the smallest source code or configuration changes in the kernel.

In this paper, we create a reference model of the control and data flow of selected representative Linux kernels. Using this model, ABI properties, and Linux's own runtime information, we apply a configuration- and instruction-set-agnostic structural matching between the reference model and the loaded kernel to obtain enough information to drive all practically relevant forensic analyses.

We implemented our approach in KATANA[1], and evaluated it against *Volatility*. KATANA is superior where no perfect profile information is available. Furthermore, we show correct functionality on an extensive set of 85 kernels with different configurations and 45 realistic snapshots taken while executing popular Linux distributions or recent versions of Android from version 8.1 to 11. Our approach translates to other CPU architectures in the Internet-of-Things (IoT) device domain such as MIPS and ARM64 as we show by analyzing a TP-Link router and a smart camera. We also successfully generalize to modified Linux kernels such as Android.

CCS CONCEPTS

• **Applied computing** → **System forensics**; • **Security and privacy** → *Operating systems security*;

KEYWORDS

memory forensics, automated profile generation, binary analysis

[1] かたな: *Japanese (single-edged) sword*

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10.
https://doi.org/10.1145/3545948.3545980

ACM Reference format:
Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. KATANA: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In *Proceedings of 25th International Symposium on Research in Attacks, Intrusions and Defenses, Limassol, Cyprus, October 26–28, 2022 (RAID 2022),* 19 pages.
https://doi.org/10.1145/3545948.3545980

1 INTRODUCTION

Memory forensics offers unique insights into the internal state of operating systems and userspace programs. Frameworks such as *Volatility* and *Rekall* have shown that it is possible to extract a large variety of information from memory dumps and have proven useful after malware and ransomware infections or to obtain disk encryption keys during governmental investigations [15].

The starting point for any such investigation is a memory dump of the target system, which can be easily obtained from virtual machines. This setting is also known as Virtual Machine Introspection (VMI). However, deriving semantic meaning from a VMI view of a virtual machine is a non-trivial challenge, commonly referred to as the *semantic gap* problem [3]. To bridge this gap, automated approaches use information collected over the lifetime of the virtual machine (e.g., instruction traces [5, 9, 23]) or rely on explicit support by the guest operating system in order to interpret data they obtain through a VMI interface [20]. On regular PCs, bare-metal servers, smartphones, and IoT devices, memory dumps can be extracted by injecting a driver or module into the running operating system or by using hardware debugging interfaces such as JTAG.

Traditional tools like *Volatility* and *Recall Forensics* do not handle forensic investigations of Linux adequately and their analyses have become outdated over time. These tools extract the necessary information about the structure of the OS using debugging information to form a *profile* containing the memory location and layout of crucial OS data structures. However, debugging information about the target is not always readily available and in those cases these tools cannot be used. Especially Linux imposes unique challenges to a forensic analyst in need of a profile: There is a plethora of kernel binaries with slightly varying behaviors that a separate profile needs to be generated for. This is caused by a myriad of compile-time configurations and the kernel's support for different compilers and compiler versions, each pursuing its own code generation strategy [25]. Consider the current definition of the

`task_struct` structure: In Linux 5.8.14, it contains 66 preprocessor directives influencing the number and position of its members.

In most commercial operating systems, on the other hand, the limited customizability means that there is often an easy path for obtaining additional debugging information. For instance, we can retrieve Windows debugging symbols from the Microsoft symbol server based on an extractable identifier from a memory dump.

In the Linux context, however, most of this information is still encoded in the guest operating system. Even if no explicit support for forensic tools is present and no debugging information is available, *the code of the system itself* still needs to be able to load and unload drivers or to examine stack frame information for crash reports, etc. Our own implementation Katana exploits this in a *code-based* approach to deduce a profile usable for forensic analysis.

Concurrent to our own work, AutoProfile [19] was proposed suggesting a similar *code-based* approach to ours. Furthermore, LogicMem [21] utilizes a *runtime information-based* approach. In contrast to *code-based* approaches analyzing the code in the `.text` segment, only the memory dump of the volatile *runtime data* of the operating system is used (e.g., the task list) in a Prolog inference system to deduct a profile. We will further discuss similarities and differences in Section 7.

The key contributions of our work are as follows:

- We provide an implementation of *code-based* profile generation and release it to the public[2]. In contrast to AutoProfile, our implementation, Katana, is based on Ghidra's intermediate representation *P-Code* and works *across architectures*. We evaluate it on x86-64, ARM64 and MIPS.
- Furthermore, we analyze how the created profile generalizes to the full Linux kernel and modern analysis plugins, while existing works (like AutoProfile and LogicMem) focus on a small set of structures for a limited set of partially outdated *Volatility* analyses.
- We prove that Memory Extraction can also be done in a *binary-only* setting by using a Linux Kernel Module (LKM), as it was possible by using the *LiME* LKM in *Volatility*.
- We perform an extensive evaluation of Katana on 85 self-compiled kernel builds with different configurations across 7 kernel versions, and perform various real-world analyses on 45 different kernels from common Linux distributions including Android. We demonstrate Katana's cross-architectural capabilities by analyzing memory dumps of MIPS-based devices (a TP-Link router and a camera), as well as an ARM memory dump.

2 BACKGROUND

In order to motivate our design decisions for the analysis framework Katana, we first discuss two mechanisms used internally by the Linux kernel to organize the mapping between symbolic names and virtual addresses. Afterwards, we explain the details of *structure layout randomization*; a relatively new (April 2017[3]) security feature of the Linux kernel. Then, we discuss why the wide variety of configuration options of the Linux kernel drastically complicates forensic analysis. Finally, we provide a quick introduction to *P-Code*; the intermediate representation we built Katana upon.

2.1 Volatility and Rekall

Volatility[4] is an open-source framework for memory forensics with support for all three major operating systems (Windows, MacOS, and Linux). In our work, we primarily refer to *Volatility* as a comparison, since it is by far the most commonly used tool, even though it was not designed to work with differing configurations and structure layout randomization. *Volatility* supports a large variety of analysis passes such as listing the currently running processes, loaded kernel modules, active file descriptors and active network connections. Additionally, analyses scanning for known rootkit artifacts and integrity checking are available. Development appears to have stalled since the most recent stable release of *Volatility* 2.6 in 2016, resulting in many of the currently existing analyses failing on recent kernel versions.

In order to function, the *Volatility* framework relies on a "profile", which contains the layout of important kernel structures and the relative location of vital global variables. The analysis plugins then utilize this information to locate and parse the kernel's internal data structures in the memory dump in order to produce their reports. On Linux, these profiles are usually generated from the kernel debugging symbols.

In 2013, *Volatility* was forked to streamline the codebase. This fork became the *Rekall Forensics*[5] framework maintained by Google, but has been abandoned now. While a few analysis plugins might work differently, *Rekall* functions in the same way as *Volatility*.

2.2 Kernel Symbol Table

The symbol table (symtab) contains the virtual addresses of *exported* functions and variables that the kernel provides to loadable kernel modules (LKMs). If an LKM wants to log a message using the exported `printk` function, a lookup in the symtab is performed at module load. As LKMs can be loaded during runtime, complete information about exported symbols must be available during runtime, a circumstance used by Katana.

Note that even for kernels compiled with *all* optional features disabled[6]—i.e., even if LKM support is disabled—the kernel still contains a symbol table. It is essential to Linux's functioning and cannot be removed. The exact layout of the kernel symbol table changed over time, but it remains easily discoverable in a memory dump. Details can be found in Figure 6 in Appendix A.

2.3 Kallsyms

Kallsyms provides another way to resolve symbol names to virtual addresses during runtime. The kernel uses it to augment backtraces with symbol names for KGDB and requires it for Ftrace, Kprobes, and other modern kernel security features like control flow integrity checking and live patching. Kallsyms-enabled kernels contain a list of kernel symbols that is extracted during compilation, compressed, and saved in the data sections of the generated Linux image.

[2] Our tools and pre-generated databases for Linux 3.7 – 5.15 are available at https://github.com/tum-itsec/katana.
[3] https://www.openwall.com/lists/kernel-hardening/2017/04/06/14
[4] https://www.volatilityfoundation.org/releases
[5] http://www.rekall-forensic.com/
[6] `make tinyconfig`

In contrast to the symtab, the kallsyms mechanism is aware of the addresses of non-exported kernel functions. If the configuration option `KALLSYMS_ALL` is enabled, it even includes names and virtual addresses of symbols that reside in the data section. As such, the number of symbols on which kallsyms may provide information is magnitudes larger than what can be learned from symtab.

Code running in the kernel can access the kallsyms system by querying one of two exported functions: `kallsyms_on_each_symbol` (introduced in 2.6.30; 2009) allows code to iterate over all symbols that are stored, while `kallsyms_lookup_name` (introduced in 2.6.4; 2004) does a name-based symbol lookup of the respective address.

This interface has been stable since its introduction, but can be disabled at compile time as it is an optional feature. Fortunately, as we will see in Section 4, most systems leave the mechanism activated, including the symbols in the data section (`KALLSYMS_ALL`).

2.4 Structure Layout Randomization

Since version 4.13, the Linux kernel has offered *structure layout randomization* as an additional security feature. If enabled, the spatial order of members of structures marked with the attribute `__randomize_layout` is shuffled at compilation time. Structures containing only function pointers will always be shuffled, if not explicitly forbidden by using `__no_randomize_layout`. The shuffling is realized as a compiler plugin, which adds an additional optimization pass to the compilation pipeline. Shuffling is implemented deterministically, based on a 256-bit random seed. This design decision enables a shuffled kernel to load kernel modules that were compiled later than the main kernel image, but has the serious drawback that distributions need to publish the random seed, making security gains in general-purpose distributions questionable.

2.5 Kernel Configuration

Because Linux targets a wide variety of use cases, it provides a large number of compile-time switches that can be used to enable, disable, and modify certain features in the kernel. Each of these *Kconfig* variables is available to both the kernel's custom *Kbuild* build system and the source code, where they allow conditional compilation of certain code fragments or source files (e.g., to configure support for specialized hardware). The same system is used to select which features are not embedded in the kernel but are provided through kernel modules, and to configure a large number of other settings ranging from the relatively benign (e.g., the default host name) to critically important (e.g., CPU endianness).

This significantly complicates any analysis of the Linux kernel, because measurements obtained on one configuration may not be valid on another. Moreover, rare combinations of configuration options may affect the system in unexpected ways or reveal subtle bugs [25]. For our purposes, the key differences between different configurations are the functions that are available for analysis and the layouts of structures in the kernel: Since many features add members related to their functionality to core kernel types, many different configuration switches can independently change the layouts of these types. This creates a vast number of possible structure layouts even when randomization is disabled. For example, the presence or absence of the 66 conditionally compiled segments in the `task_struct` structure on Linux 5.8.14 depends on 56 different Kconfig variables.

2.6 P-Code

P-Code is an intermediate representation specifically designed for reverse engineering applications and is implemented by the popular software reverse engineering suite *Ghidra*[7].

Processor-specific instructions are lifted to a corresponding sequence of P-Code operations. This means that analyses built on top of P-Code are architecture-agnostic, as long as the lifting process is implemented for the architecture in question. Currently, Ghidra has support for a broad range of architectures (the current version claims to support 77 architecture variants), including the most popular ones such as x86-64, MIPS, ARM, Sparc, PowerPC, etc.

Another advantage of P-Code is that it greatly simplifies implementation of higher-level analysis: The number of different P-Code operations is limited to around 60, while CISC instruction sets such as x86-64 are much more complex with somewhere between 1000 and 4000 distinct instructions depending on the method of counting.

Ghidra itself uses P-Code internally to implement many of its analyses, the most notable being its decompiler.

3 KATANA

In the following, we discuss a new, binary-only, fully automated approach for performing forensic memory analysis.

3.1 Design Goals

When we built Katana, we placed special emphasis on the provision of forensic analysis capabilities in a post-mortem, binary-only, automated, and robust fashion. Below, we briefly explain each of our design goals.

Post-Mortem Availability In many cases, it is desirable to monitor a production system that has not previously been prepared for forensic analysis. To this extent, our analysis approach needs to be able to operate on a physical memory snapshot containing a vanilla Linux kernel and user space without the presence of special debugging information, the respective kernel configuration, or the `System.map` file. A snapshot of the architectural state of the CPU (containing model-specific registers and control registers) can be present to speed up analysis, but is not strictly required.

Katana supports many different sources of memory snapshots, whether acquired using physical tools (e.g., JTAG), using hypervisor support (e.g., to analyze a compromised VM), or directly from the target system (e.g., `/proc/kcore`). For situations in which we would need Katana's output in order to obtain a memory dump in the first place (e.g., IoT devices without JTAG ports), we provide a kernel-mode memory dumping utility (UDM) that can be used without detailed knowledge about the target system (cf. Section 3.7).

Binary Only All information about the system should be derived from the compiled kernel and the respective data structures. We do not require the source code or the build toolchain of the kernel at analysis time, even if some custom kernel patches are applied. Of course, there are some limits to how many changes Katana will be able to accommodate, but commonly used kernels with such modifications (e.g., Android) are supported out-of-the-box. This

[7] https://ghidra-sre.org/

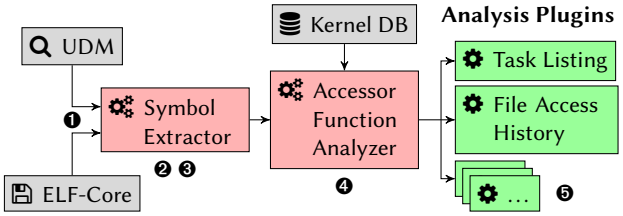

Figure 1: Design of Katana

eases the analysis of embedded devices such as Wi-Fi routers and IoT cameras that do not use "standard" off-the-shelf Linux distributions, where manufacturers may be hesitant to provide source code or debugging information even upon request.

Robust Automated Layout Derivation Katana should be capable of finding all required data structures on its own, without the analyst having to provide external information, such as the kernel version or the layout of core data structures by means of a profile-like database. This includes information such as the virtual and physical KASLR bases, as well as the binary layout of randomized kernel structures. This property offers a highly valuable complement to existing analysis frameworks such as *Rekall* and *Volatility*, as both struggle with the presence of KASLR, structure layout randomization, or configuration changes that propagate to the binary layout of the Linux kernel. The necessary maintenance effort is minimized as Katana adjusts to changes in the kernel source code and configuration without manual intervention, which still remains the predominant method in existing tools to solve this problem [2].

3.2 Overview

Katana operates in four core steps illustrated in Figure 1. Before analysis ❶, a memory dump must be obtained from a virtual machine or a bare-metal device (e.g., via our UDM or as ELF-Core file). ❷ We scan the virtual address space for the kernel's symbol table to obtain a list of functions and their respective virtual addresses. ❸ Using this list, we invoke the `kallsyms_on_each_symbol` iterator function using the popular *Unicorn* emulator [18] to obtain a more complete list of symbols provided by the kernel and all loaded modules. ❹ We match the code within the memory snapshot against a pre-generated database of *accessor functions*, which are known to access or modify specific members of data structures. This matching step is made architecture and instruction encoding independent by first lifting the identified function to P-Code and processing the resulting operation sequence. Suitable candidates for *accessor functions* and information about *invariant members* are generated *a priori* using a custom analysis plugin for the GNU C compiler (GCC) that observes the compilation of a reference kernel. We distribute pre-generated databases for all kernels between versions 3.7 and 5.15 and some older kernels alongside Katana (depicted as *Kernel DB* in Figure 1).

Katana infers the layout of frequently used kernel structures in a completely automated fashion based on the analysis results of the GCC plugin. The analysis is remarkably robust in an overwhelming number of cases, as the most important types are used at various locations in the kernel resulting in a large number of potential accessor functions. Should the analysis fail to obtain the structure layout from any function, it can easily recover using the remaining functions.

The output of step ❹ is called *profile* and could be used to drive different kinds of forensic analysis tasks, e.g., listing all running processes of a system. Katana has its own set of analysis plugins (❺), but the *profile* could also be converted to drive analyses implemented in the *Volatility* or *Rekall* frameworks.

3.3 Database Generation

To find the structure members we are interested in, we introduce the notion of an *accessor function*. As the name indicates, this is a kernel function that *accesses* a certain structure member. Note that while *pure* accessor functions, which only have the *designated* purpose of returning the value of a certain struct member, are a rather uncommon programming construct in the Linux kernel, there are plenty of functions which just *access* the desired data. For example, the `send_sig_all` function in the kernel enumerates all processes while sending a signal to each process. Katana can make use of this fact to derive the location of the process list, sidestepping the *actual* signaling purpose of the function. As such, a myriad of functions can serve as accessor functions in the context of our work. We generate a mapping between accessed structure members and accessing functions using a GCC compiler plugin.

It is important to note that basing the analysis on the output of a GCC compiler plugin does *not* contradict the binary-only design goal of our work. In fact, the source code analysis needs to be performed only *once* on a kernel version reasonably close to that of the target memory dump and can then be reused for other memory dumps. Version and configuration mismatches are only an issue if the code base of the relevant accessor functions changes significantly. Otherwise, such a mismatch may mean missing out on *additional* information that would have been present if changes are taken into account, or a few of the results of the GCC plugin becoming invalid. Neither will significantly worsen the analysis results. We can further improve accuracy by selecting the database matching to the Linux version, which can be extracted from the memory dump without the use of accessor functions. We evaluate the impact of configuration differences in Section 4.

GCC's plugin system allows inserting arbitrary transformation and optimization passes between those that are performed by default. We insert a custom non-modifying compiler pass behind the *einline* pass, at which point many short functions that will always be inlined (e.g., those marked in code with the `always_inline` attribute) are already inlined. At this stage, GCC represents each function using an intermediate language called *GIMPLE*. This representation still closely resembles the structure of the original code, albeit transformed into static single assignment (SSA) form.

We locate accesses of structure members by recursively walking the operand trees in each of the GIMPLE statements and searching for `COMPONENT_REF` (member access) and `MEM_REF` (dereference) nodes. The context in which each node appears reveals how the structure member is used. For example, if the accessed member is passed into a function call, the function argument will refer to an SSA node.

Performing a dataflow analysis across the SSA assignments yields a `COMPONENT_REF` node that refers to the member in question.

In order to support many different kernel versions with different required minimum/maximum GCC versions, the plugin supports any GCC version starting with version 4.8. Using this setup, we generated databases of structure accesses for Linux kernels starting at version 2.6.33, although earlier kernels may also be supported.

3.4 Obtaining Kernel Symbols

With any memory snapshot, the first step is to map virtual addresses to physical memory. If the memory dump already comes with page table information, or if it contains the location of the page tables in register data (such as on x86 via the CR3 register), this is straightforward. Otherwise, we can identify candidate page tables in memory by their recursive structure, validate them based on architecture-specific constraints, and find the kernel page table by choosing the candidate with the largest number of valid kernel memory mappings. Memory dumps created by our toolkit (c.f. Section 3.7) already contain paging information.

Reading the Kernel Symbol Table Armed with the virtual to physical mappings, we scan the virtual address range for the location of the Linux kernel symbol table (.ksymtab; described in more detail in Appendix A). By using the `.ksymtab` data, KATANA obtains the locations of exported function starts and global variables in the memory dump.

Augmenting the Symbol List Using Kallsyms We are now equipped with the names and addresses of all symbols the Linux kernel exports using the `EXPORT_SYMBOL` macro. This list can be substantially augmented using additional information provided by the *kallsyms* mechanism.

The `kallsyms_on_each_symbol` function has been present in the kernel since version 2.6.30. It iterates through all symbols known to the *kallsyms* mechanism. For each iteration, this function transfers control to a user-provided callback function. With the information contained in the symbol table, KATANA is able to find the location of `kallsyms_on_each_symbol` in virtual memory and to subsequently invoke it in the context of the memory snapshot. To be tolerant to implementation changes, such as the presence of the `KALLSYMS_RELATIVE_BASE` configuration flag, we execute this function in the *Unicorn* CPU emulator. Inside the emulator, we call `kallsyms_on_each_symbol` with a prepared callback function as a parameter. The emulated code will in turn hand control to the callback function for every entry present in the *kallsyms* database. This allows us to obtain a list of memory locations of all non-static Linux kernel functions. In the event that `KALLSYMS_ALL` is enabled, we are also able to retrieve the addresses of symbols residing in the data segment of the kernel.

In the unlikely case that *kallsyms* is fully disabled in the analysis target, it is necessary to solve the function identification problem using another approach. On its own, the information derived from only the symbol table is insufficient to obtain good results. However, this is not an issue in practice: *kallsyms* is enabled on *all* systems we analyzed (including resource-constrained IoT devices), and a required dependency for other kernel features (cf. Subsection 2.3).

Because using the kallsyms API allows modules to circumvent licensing restrictions in the kernel, the `kallsyms_on_each_symbol` function is no longer exported (but still available internally) starting with Linux 5.7. In this case, we recover its location using the related *kprobes* API, which is ordinarily used to place arbitrary breakpoints in kernel code and returns the address where the breakpoint has been set.

3.5 Automated Structure Layout Reconstruction

In order to reason about the contents of the concrete memory dump, we have to infer the offsets of structure members by matching the machine code of the accessor functions with our pregenerated *Kernel DB*. The matching process uses several properties maintained during the compilation process: First, each accessor function, when not inlined, has to obey the respective architecture-dependent Application Binary Interface (ABI). Second, certain pointer calculations, such as the `container_of` macro, and the use of global variables result in recognizable instruction patterns.

We derived the following analyses from these observations:

ABI to caller function How arguments are passed to a function is defined by the ABI[8]. During analysis, KATANA will exploit this by tainting incoming arguments and tracking every access that happens relative to a tainted register. In the example depicted in Figure 2, we track the value of the `rdi` register (or its P-Code equivalent). Note that its value is first moved to `rbx`, whose value is preserved across function calls (as specified by the ABI). Our taint analysis would now propagate the tainted property to the assigned register (i.e., `rbx`). Then, the actual dereferencing operation happens in line 6 (①).

ABI to callee functions Similarly, calls to non-inlined functions must follow the respective ABI. Therefore, we resolve the address of every called function (e.g., `printk` in Figure 2) inside the function body to its name. If our static analysis of the C code observed that a call to this function contains relevant arguments (i.e., a field access), KATANA tracks the affected arguments backwards by following the observed data flow (e.g., across simple assignments). Inside this slice, we find the last indirect memory access, and consider the displacement used in that instruction as a potential offset for the target field. For example, we can see this occurring in Figure 2, when `t->pid` (②) is passed to `printk`.

ABI from callee functions The return value location is specified by the respective ABI as well. Similar to the way we tracked accesses to parameters, we track pointers that are being returned by tainting the memory location that contains the return value after a successful function call.

ABI from caller function Just as functions that are being called must return their result in a predetermined location, the function we are currently analyzing has to as well. If the currently analyzed function returns a struct member, we will follow the data flow backwards to identify the last indirect memory access that affected the return register. In our imaginary function `foobar`, the pid member of *t* is returned (④). The data flow analysis shows that the value in `eax` (the ABI-specified return register) comes from an access with relative offset 0x3e0.

[8]For example, on x86-64, the System V ABI mandates that the first six arguments of a function have to be passed in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`.

Source code:

```
int foobar(struct task_struct *t)
{
  printk(KERN_DEBUG "Hey!\n");
  if(t->mm ① != NULL) {
    printk(KERN_INFO "PID: %d IPID: %d\n", t->
     pid ②, init_task->pid ③);
  }
  return t->pid ④;
}
```

x86-64 disassembly (Intel syntax):

```
push   rbx
mov    rbx,rdi
mov    rdi,0xffffffff821fb5d4 # "Hey!"
call   ffffffff810b9229 # <printk>
cmp    QWORD PTR [rbx+0x3e0]①,0x0
je     out
mov    esi,DWORD PTR [rbx+0x490]②
mov    edx,DWORD PTR [rip+0x131d6fb]③ # <init_task+0x490>
mov    rdi,0xffffffff821fb5dc # "PID: %d..."
call   ffffffff810b9229 # <printk>
out:
mov    eax,DWORD PTR [rbx+0x3e0]④
pop    rbx
ret
```

❶ Access via calling function parameter
❷ Access via ABI to callee functions
❸ Access via global properties
❹ Return of a local member
● Visible structure offsets
● Composed offset

Figure 2: Structure accesses in an example piece of C code, and the result of compiling them to assembly using GCC 9.

Access to globals Global variables in the kernel have to be accessed either relative to the current instruction pointer or through the variable's absolute address. While scanning a function, whenever such a reference to the data segment is encountered, we obtain the closest known symbol location before that address and treat the difference (within some reasonable limits) as the offset for the respective field. From the information stored in our database, we infer *which* of the fields was accessed. In Figure 2, `foobar` accesses the `pid` member of the global variable `init_task` (③). This pattern also allows us to deduce the locations of some missing globals if `KALLSYMS_ALL` is disabled.

The `container_of` macro To implement features such as linked lists and hashtables, the Linux kernel often uses the `container_of` macro to access parent objects containing other objects. For example, `task_struct` objects contain the member `tasks` of type `list_head`. In turn, `list_head` contains a member `next`, pointing to the next object in the list. Typically, one would expect `next` to directly point to the following `task_struct` object, however, it actually points to the `tasks` member inside the next `task_struct`. This allows the kernel to implement code that traverses these lists of type without depending on the offset of the `list_head` inside each of the stored objects. However, to access the actual object referenced by the list, the pointer to the next list entry needs to be adjusted by that offset, which is done in kernel sources by using the `container_of` macro.

We observed that in a large majority of cases, Ghidra will emit an INT_ADD P-Code operation with a negative immediate as opposed to a more natural INT_SUB operation with positive immediate. This Ghidra-specific heuristic is consistent across architectures in version 10.0.4 of Ghidra upon which we evaluated KATANA. As we observed similar patterns in the underlying machine code, we believe this pattern will also be present in the lifting behavior of future versions. During analysis, we collect these unusual arithmetic operations and attempt to match them with known uses of the `container_of` macro, which we detect in the AST with our GCC plugin. Then, the displacement is likely the inverse of the offset of the member referenced by the `container_of` macro.

Invariant members All techniques listed so far are driven by KATANA's ability to identify and exploit ABI features of the machine code during the invocation of a function. However, there are some important members, which are accessed only inside a long call chain of multiple static functions that are inlined into the calling function. Due to additional compiler optimizations, these functions are usually not matchable to the original dataflow for KATANA. Instead, these members can be partially recovered by relying on a database of *invariant members*, i.e., structures that are *not at all* covered by `#ifdefs` (and therefore do not change with configuration changes) and are *not* marked for randomization. We extract these members using our compiler plugin and use them in our majority voting process with a double vote, but they can still be overruled given sufficient contradictory evidence.

Specifics of P-Code and the overall algorithm As mentioned earlier, KATANA's structural matching is performed on P-Code, which we acquire by accessing the Ghidra API. The classic `[reg+offset*mul]` instruction pattern on x86-64 seen in Figure 2 is decomposed into several P-Code instructions. We track P-Code's equivalent to registers and memory locations (so-called "varnodes") and emulate mathematical calculations on constants in order to accurately follow the data flow. On every `LOAD` and `STORE` operation, we determine if we can match the accessed object to our database. Using P-Code also makes our analysis agnostic to compiler specifics, because the semantics of the lifted representations remain the same and will also be matched by our algorithm. Further details about the lifting process can be found in Appendix C.

Our analyses are complicated by the fact that the compiler applies various optimizations during compilation. The compiler is free to reorder operations, as long as the data flow permits it, to inline functions, and to eliminate dead code. We do not match conditions between the source code and the machine code, meaning that each time the control flow branches, inaccuracies may be introduced. To address this, we perform a weighted majority voting on all offsets recovered for a single struct member. The weights are applied based on the type of analysis that the offset was recovered from and reflect correctness probabilities that we empirically obtained from

Algorithm 1: P-Code to source code database matching

```
S ← recovered function symbols;
D ← access database for similar kernel version;
O ← ∅, mapping of members to offsets;
foreach recovered function f ∈ S do
    A ← ∅, ordered set of accesses;
    foreach P-Code instruction i ∈ f do
        if i is a structure member access on some o then
            ▷ Recover source or sink s and offset δ
            s, δ ← taint-tracking(o);
            A ← A ∪ {(s, δ)};
        end
        if i is a register access with fixed offset δ < 0 and
          |δ| < |δ_max| then
            A ← A ∪ {(containerof, −δ)};
        end
    end
    foreach P-Code analysis λ do
        R ← reference accesses from database D(f, λ);
        foreach matching access a ∈ λ(A) do
            ▷ Fetch type t and member m
            t, m ← next(R);
            O(t, m) ← O(t, m) ∪ {offset(a)};
        end
    end
end
```

analyzing the 85 kernels displayed in Table 3. Together with this majority voting, we will see in Section 4 that these transformations do not harm our analysis in almost all cases if the field inside the structure is accessed frequently enough. Katana may pick an incorrect candidate for a struct offset in a single accessor function, but can correctly vote out the final offset.

A broad overview of our P-Code-to-database matching is depicted in Algorithm 1. We sequentially iterate over the P-Code instruction stream of every recovered function, following direct branches and analyzing both control flows of conditional branches, combining their results and removing duplicates. When we encounter an instruction that could conceivably be a structure member access, we follow our description of the taint tracking information both forwards (where the result of the member access passes to data sinks such as function call arguments) and backwards (to data sources such as global variables and function parameters) as described above, and log the sink or source and the recovered offset in an ordered list of accesses. Candidate accesses using the `container_of` macro are treated similarly, but no taint tracking is necessary. After obtaining the set of accesses for a function, we separately consider each of our analysis passes. Reference accesses from the database (containing type and member information) for that specific analysis pass are matched one-by-one to the accesses obtained from P-Code (which contribute the recovered offset).

3.6 Analysis Plugins

In total, *Volatility* implements 66 different analyses for Linux memory dumps. However, 14 analyses either do not work correctly as per *Volatility's* own source-code comments (e.g., `linux_arp`) or work only on extremely outdated Linux versions. Therefore, creating a profile for these analyses and executing them would not allow for a fair evaluation on modern kernels. Instead, we identified and reimplemented a set of important analyses found in existing work (e.g., in Ligh et al. [15] and in *Volatility*) and include them directly into the Katana framework. This design decision also allows us to enable fallback access patterns for complex analyses and to avoid using excessive structure accesses where it is not needed to fulfill the analysis task. These excessive structure accesses tend to happen in *Volatility*'s analyses as its profiles are always correct. Unlike for Katana, relying on unnecessary structure accesses therefore does not increase the probability that the whole plugin fails.

We have reimplemented the following analyses in Katana:

- We obtain the current kernel version *banner* from a global symbol.
- We extract the kernel *dmesg* ringbuffer logging output from the memory dump. The corresponding Volatility plugin is an example of an outdated plugin, because the internal structures have changed in version 5.10 (released in December 2020) and *Volatility* has not adapted its analysis.
- We derive the list of *modules* currently loaded in the kernel.
- We obtain a *process listing* from the snapshot. This includes both the process names, execution state, and the user ID with whose permissions the process is running. From there, we use the memory mappings stored by the kernel to produce ELF core files containing the virtual address space of each userspace process. This allows us to match memory segments in the snapshot to their respective processes. The resulting core files can then be analyzed using classical reverse engineering, debugging, or forensic analysis tools.
- Based on the process list information, we retrieve information about *environment* variables, a list of currently *opened files*, and currently open sockets and active *network* connections akin to the `netstat` command. Using this information, an analyst can quite accurately reconstruct the state of userspace processes at the time the snapshot was taken.
- We also provide the ability to list all network neighbor tables including the *ARP table*. This allows the analyst to reconstruct a list of IPv4 and IPv6 machines the analysis target was communicating with at snapshot time.
- An analysis walking through the kernel heap's set of allocated and formerly allocated objects is included. Of particular interest is the heap cache containing `dentry` (*d*irectory *entry*) objects. We associated each of these objects with corresponding metadata, which allowed us to build a timeline of file accesses. *Volatility*'s's's `dentry_cache` plugin performs essentially the same analysis, but has not been updated since 2014 and requires a global symbol that was removed with Linux 3.6 (dated September 2012). Therefore, it operates only on kernels that use the SLAB allocator, while we also support newer kernels with the now-default SLUB allocator. Furthermore, additional security features (i.e., pointer mangling of

free list pointers) have been developed since then and are, therefore, not supported.
This plugin is one example requiring the knowledge of many kernel structures, where our own implementation utilizes an alternative access strategy if a key structure offset could not be recovered.

3.7 Module-based Snapshot Creation

While capturing a memory dump from a virtual machine or via hardware debugging primitives such as JTAG should generally be preferred, these approaches are often highly specific to each device or virtualization environment, and not all devices expose the necessary interfaces for this kind of access.

For analyzing Android devices, for example, it has been an established practice to insert a kernel module to dump the system memory from within the same privilege domain as the operating system. One frequently used solution that takes this approach is the *LiME* toolkit [15, p. 580].

However, building a kernel module generally requires access to the kernel headers, the kernel configuration, and the seed used in structure layout randomization, none of which are available in a *binary-only* analysis setting. Unfortunately, it is also not feasible to precompile *LiME* for the different kernels one may encounter: Linux kernel modules are generally compatible only with the specific version of the kernel for which they were compiled.

To allow also for memory snapshot acquisition in a binary-only setting, the Katana framework includes a module that adapts dynamically to the targeted kernel version. The snapshotting processes is described in the following. First, a custom loader and the LKM are compiled for the target architecture (currently x86-64, ARM64, and 32-bit MIPS systems are supported). Second, the loader and the LKM are transferred to the analysis target. Afterward, the loader will analyze the existing kernel modules on the system and alter the `.modinfo` section of the LKM accordingly, parameterize the LKM with the exact Linux version, and issue the insmod system call to insert the LKM. The LKM is designed to avoid direct accesses to structure members that are influenced by kernel configuration options, because their offsets are still unknown at this point in time. Where APIs have changed over the years, we use the injected kernel version information to choose between alternative implementations.

In a last step, our module sends a full memory dump of any supported system to a server on the local network, including the full page table of the CPU on which the memory dump is taken. The server can then carry out the snapshot analysis steps of Katana. Our kernel module is loadable on any Linux kernel since version 2.6.18 (dated September 2006) and enables us to obtain memory snapshots in situations where other tools may not be usable. These situations could arise when the device does not allow direct physical exfiltration, e.g., with an Android phone or a bare-metal server. Such devices usually do not offer debugging interfaces like JTAG.

4 LAB EVALUATION: COMPARISON TO VOLATILITY

For the first part of our evaluation, we create lab conditions (i.e., kernels with debug symbols for ground truth) such that we can quantitatively evaluate the performance of Katana. In consequence, this experiment illustrates the magnitude of the impact of varying kernel structures in practice compared to profiles created heuristically by Katana.

To conduct this experiment, we first assess the accuracy of the structure layouts recovered by Katana in the presence of different randomized kernel configurations, and compare the results with the fields required by *Volatility*'s's for its various analyses. We utilize Volatility's own analyses here for a fair comparison, but exclude 14 analyses that do not work correctly (see Section 3.6).

Experiment Setup To evaluate our automated structure layout recovery, we analyzed 85 different builds of 7 kernel versions and compared the recovered member offsets with the debugging symbols for these kernels. More specifically, to cover a large range of kernel versions, we chose to evaluate Katana on five current long-term support versions, an older, now unsupported, LTS version, and a recent stable kernel version. For each kernel version, we generate a reference database for Katana based on that kernel version's default configuration (*defconfig*). Afterwards, we compile additional variants of each kernel version with modified configurations, and examine how well the reference database generalizes to the changes.

To generate test kernels, we make use of Kbuild's *randconfig* option, which randomizes the features that are enabled in the kernel. While some of these kernels may not be bootable, this produces a set of unbiased configurations, in which features and their corresponding data structures differ from the reference kernel. Using this process, we generate 10 different configurations of varying size on each of the 7 kernel versions we investigated. Support for x86-64 with SMP multithreading, `printk`, and kernel module support is forcibly enabled on all kernels. Where supported, we generate two kernels with the default configuration and structure layout randomization enabled. Furthermore, we save ground truth structure layout information for all kernels. In order to avoid booting each of the 85 kernels — after all, some of them might not even be bootable — we let Katana extract structure offsets from the `.text` sections of the unbooted kernel images.

We identify the structures and members required by *Volatility*'s analyses by inspecting their implementation. For these members, we compare the structure offsets that Katana extracted from the kernel images against the ground truth information. Since Volatility uses the offsets of the reference profile to power the 52 analysis plugins, we have to consider a single differing offset in the target kernel as leading to analysis failure.

Results Across the 70 different kernels with randomized configurations, we correctly recover between 65.4% and 79.3% (average: 73.58%) of all members identified by the reference database. This also includes fields used only internally by drivers or other specialized code that may not even be active on the target system and that would usually be irrelevant for forensic analysis. However, it would be incorrect to exclude these members from the statistical evaluation ahead of time, because they may become interesting for

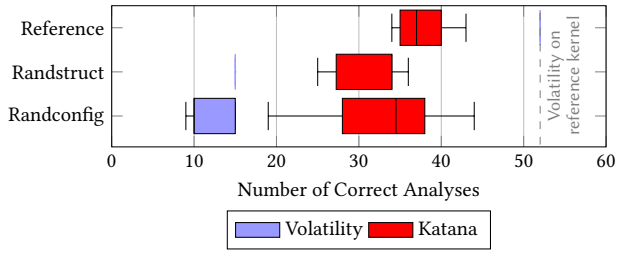

Figure 3: Working Volatility analysis plugins using Volatility and KATANA offsets

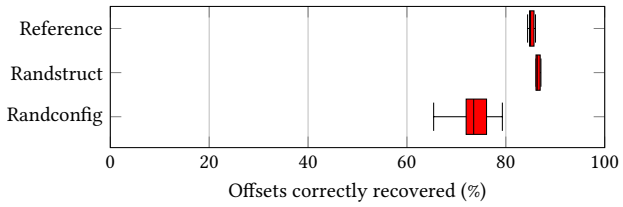

Figure 4: Fraction of correctly recovered structure member offsets using KATANA (all kernel structures)

future analyses, and a manual review of all members is impossible. On average, wrong offsets are recovered by KATANA for 15.14% of members (8.1% – 20.8%), and no value can be recovered for the remaining 11.28% of members (6.4% – 16.9%).

Due to the structure differences introduced by the configuration changes, *Volatility*'s is able to successfully perform only 9 to 15 of the 52 analyses (median: 10), while KATANA correctly recovers enough offsets to perform between 19 and 44 analyses (median: 34.5).

If structure layout randomization is enabled, KATANA's performance improves further: We now correctly recover an average of 86.46% of known offsets. Again, KATANA significantly outperforms Volatility; we manage to run between 25 and 36 analyses (median: 34), while *Volatility* can only perform exactly 15 on each of the 8 tested kernels.

On the reference kernels for each version, *Volatility*'s (with perfect debugging information) is able to run all 52 analyses. KATANA correctly recovers the offsets of 85.17% of known members on average, and can perform between 34 and 43 of the 52 analyses. Figures 3 and 4 summarize these results; details can be found in Table 3, including the total number of members to which offset recovery relates[9].

In general, we can see that, when ground truth information is not available, KATANA outperforms *Volatility* with regard to the number of correctly performed analyses. Often, KATANA can still execute central analyses including listing processes (`linux_pslist`, supported 41x by KATANA, but by Volatility only in the 7 reference kernels) and environment (`linux_psenv`, 68x vs. 7x), network connections (`linux_netstat`, 66x vs. 7x), and module listings (`linux_lsmod` 78x vs. 42x). Where KATANA fails to perform *Volatility*'s analyses, this is most frequently caused by the architectural differences between the two: *Volatility* generally opts to require more structure members than strictly necessary for the analysis (e.g., `linux_check_syscall` verifies the integrity of the system call table, but needs to recover a file from memory in order to do so, even though both syscall numbers and the target pointers are known ahead-of-time), and 11 of the plugins perform integrity checking with the goal of detecting the presence of malware (e.g., `linux_check_inline_kernel`), which requires a particularly large number of members. Since analyses are all-or-nothing, i.e., a single misidentified member leads to analysis failure, there is a significant threshold effect: slight differences in recovery can lead to large differences in the number of successfully performed analyses. This can be observed, e.g., in the kernels with structure layout randomization for kernels 5.8.14 and 5.4.70.

[9]Note that since kernels with structure layout randomization contain *un-randomized* debugging information, we instead report the number of members as observed by our GCC plugin during compilation, which for Linux 4.19 and 4.14 differs significantly from the set of members reported by debugging information.

5 REAL-WORLD EVALUATION

To evaluate KATANA in real-world usage scenarios, we cannot rely on the availability of ground-truth information. Instead, we executed all of our implemented analyses (see Section 3.6) on a set of test systems and manually validated the correctness of the output. The set contains a variety of popular Linux distributions, snapshots from non-x86 architectures, a VM infected with malware, and off-the-shelf IoT devices. Our results are described in the following sections.

5.1 Linux Distributions and Variants

To obtain a comprehensive overview of common Linux kernel configurations, we collected a broad variety of popular Linux distributions and variants including Android, with 45 kernels ranging from version 3.9.5 (June 2013) to 5.11.16 (May 2021). We generate snapshots of fully booted QEMU virtual machines using QEMU's `dump-guest-memory` command and analyzed them with all analysis plugins we implemented.

We found that the kallsyms feature is enabled on all of the distributions and release versions examined, and that `KALLSYMS_ALL` (adding global variables and non-exported functions to the set of symbols available through kallsyms) is enabled on all systems except for Debian Jessie and Android 9.

We were able to successfully recover the kernel version *banner*, *dmesg* ringbuffer contents, and the full *module* listing on all memory dumps except for Android 11, where we mis-predicted an offset. In all other cases, KATANA recovered the offsets for the `module` struct, and the address of the internal linkage symbol `modules` successfully. On the other three Android snapshots, KATANA correctly recovered the offsets in the `module` struct, but no modules were loaded.

Our more complex analyses were able to automatically produce task listings and core dumps for the userspace processes as well as the list of open file descriptors for all 45 images. We verified that the generated core dumps matched the memory maps reported by the `/proc` entry for that process on the virtual machines themselves.

Enumerating the processes' environment variables succeeded on all but two images (where empty environment strings were reported). Network-related analyses appeared to be more fragile: listing the ARP table and network connection information both succeeded in 35 of the images. Our analysis of the kernel cache to discover a history of file accesses succeeded on all but seven images taken from standard Linux distributions, but failed on all taken from Android systems.

A listing of the members used by our analysis passes can be found in Appendix B. Furthermore, a detailed listing of all Linux systems showing which of our analyses work can be found in Table 2.

5.2 IoT Devices

Katana is specifically designed to also support forensic analysis of IoT devices. IoT devices are by nature restricted to binary-only approaches, since many vendors do not publish debugging symbols, build configurations, or modifications to the kernel source. We therefore evaluate Katana on two MIPS-based physical devices and one ARM64-based VM (results are also included in Table 2). All analyses were performed cross-architecture against a database generated on x86-64.

Katana was able to perform all of our analyses except the file access history on the memory dump from an ARM64 VM running Linux 4.19[10]. On a TP-Link *TL-WR740N* router and a *Tapo C200* smart camera, Katana managed to recover all data necessary to correctly extract both system (modules, dmesg log, etc.) and process information (including environment variables and open file descriptors). Missing offsets for some rarely used members prevented us from running some of the more complex analyses (e.g., the file access history). The router contains a MIPS 32-bit big endian processor running an extremely outdated Linux 2.6.31; the camera runs Linux 3.10 on a little-endian MIPS 32-bit chip. We attached to both devices using UART and produced memory dumps using Katana's custom dumper.

5.3 Real-World Malware Analysis

In order to prove Katana's practical utility in a realistic post-compromise scenario (malware infection with persistence), we deployed a sample[11] of the RedXOR malware [13], which we randomly selected from VirusShare, on an isolated virtual machine (Ubuntu 18.04, Linux 4.15) and analyzed a memory snapshot using Katana. We were able to successfully recover the malware (and its process image) from the list of processes. From the memory dump, Katana also obtained a timeline of file accesses and the list of files currently open at the time the snapshot was taken (revealing the source of the infection and its persistence mechanism), and observed that there were no currently open network connections or sockets. Our results (obtained in a post-mortem setting) closely matched those obtained both by malware analysis frameworks using live introspection, and manual analysis [13]. Detailed results can be found in Appendix D.

[10] At the time of evaluation, QEMU did not implement dumping memory of ARM64 guests, so we deferred to using Katana's memory dumper instead (Appendix 3.7).
[11] SHA256: `4f159f6a745752e3211ca1146830c86075fd8f5db60f704605a57db904dcf5c5`

5.4 Performance

We evaluated Katana's performance on the real-world snapshots from Section 5.1. Overall, Katana performs fast enough to be used on a daily basis by an analyst. We exclude the IoT device snapshots from the data below due to their significantly smaller size.

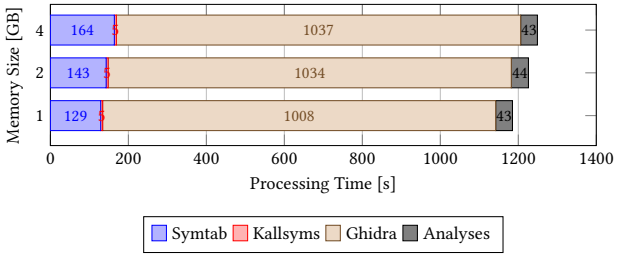

Figure 5: Performance on differently sized memory snapshots

Database Generation During our evaluation, we did not observe a noticeable impact of our GCC plugin on kernel compilation times. This means that creating the database of accessor functions is approximately as fast as normal *defconfig* kernel build (about five minutes on our hardware, depending on the kernel version), and can be performed using multiple cores. We used an AMD Ryzen 2950X with 32GB of RAM and compiled the kernel with eight parallel threads (`-j8`). During normal analysis, this database will be precomputed in almost all cases; we will distribute databases for kernels 3.7 through 5.12 alongside Katana.

Layout Reconstruction Locating the symbol table took between 32 and 193 seconds depending on the symbol table layout, for an average of 72 seconds. Emulating kallsyms takes between 1.5 and 6.1 seconds; the impact on end-to-end performance is negligible. Recovering the structure layout using Ghidra takes the largest amount of time. On average, processing any of the 45 non-IoT snapshots listed in Table 2 took 883 seconds, with values between 10.5 and 18.5 minutes.

Analyses The time taken for further analysis is generally negligible. All analyses, except those for which userspace paging needs to be reconstructed (environment variables, and process core dump creation), finished within six seconds on all 45 non-IoT snapshots. This matches the speed of comparable *Volatility* analyses. The file access history had both the highest average and maximum runtime, at 2.4 and 6.2 seconds, respectively. The performance impact of recreating userspace paging greatly depends on the amount of memory that needs to be mapped and its page table layout. For example, extracting 4.5GB of core files across 107 processes from the Android 10 snapshot took 169s, while memory dumps with fewer processes were much faster (on average, 32 seconds for 470MB of output).

Impact of Snapshot Size The size of the memory dump itself does not significantly affect the end-to-end analysis time. This is due to the fact that the analysis is strongly affected by *what* data is in the snapshot, rather than *how much*. Figure 5 compares the average performance of each of the layout reconstruction steps for otherwise identical memory dumps of different sizes (averaged

over multiple dumps). We see that the overall performance is essentially independent of the size of the snapshot. This is expected for kallsyms emulation and later steps (the size of the input to these steps is bounded by the specifics of the kernel in question, rather than the amount of kernel data). The symbol table search does scale with memory size, but only slightly — more time is spent validating matches than actually scanning the memory.

6 DISCUSSION

Using P-Code allows Katana to target a wide range of devices when an appropriate P-Code implementation is available. We showed its effectiveness for analyzing targets such as Linux distributions (cf. Section 5.1) and IoT devices (cf. Section 5.2). Even a mismatch between the architecture of the generated database of accessor functions and the running kernel still allowed Katana to provide valuable forensic information. In case Katana produces false positives, an analyst should easily be able to tell them apart from accurate information. During our evaluation, we only encountered results that could be misinterpreted as a false negative once: the empty module list on Android kernels.

6.1 Design Decisions and their Impact

In previous iterations of Katana, we based our analysis implementation on the *Capstone* disassembler and its metadata. During the transition to P-Code, we observed a general increase in accuracy of our analyses when compared to the assembly-based approach. Ghidra's ability to derive higher-level meaning from machine code during the lifting process (a feature heavily relied upon by Ghidra's decompiler) greatly improved the quality of our results, and we profited from out-of-the-box support for multiple architectures.

Preserving additional information on *invariant members* and injecting it into the majority vote provides a significant improvement in the recovery rate compared to only using accessor functions. In essence, this combines the *Volatility* approach of assuming offsets never change (which we can guarantee in the case of invariant members as long as the kernel was not modified) with the accessor function approach (which allows us to overrule the former in case our assumptions are not correct). This maintains most of the flexibility of the latter while still benefiting from ground-truth information.

We decided to re-implement some of *Volatility*'s analyses with Katana in order to update the analyses to work with recent kernels and to reduce dependence on technically optional structure members. *Volatility* can assume these are always known because of perfect profiles, but this makes it significantly more difficult for Katana to fully drive *Volatility*'s analyses in cases where almost all, but not all members are recovered correctly.

Instead of relying on ABI characteristics for matching, another naïve approach would be to utilize *BinDiff* to perform an instruction-to-instruction matching at binary level and match the changed offsets to structure members. However, this is particularly sensitive to changes in optimization level or compiler version and to larger code changes (e.g., `#ifdefs`). We found that even if we let *BinDiff* utilize kallsyms information for function identification, a precise instruction-to-instruction matching cannot be performed in the majority of cases (e.g., Figure 9), and mapping instructions to structure members remains a difficult task. Even with debugging information available, DWARF can only map address ranges to line ranges, which is a too coarse to identify individual accesses.

6.2 Rootkit Resilience

Finally, we discuss our ideas in the context of Direct Kernel Structure Manipulation (DKSM) [1]. This usually necessitates a lengthy discourse on the presence of rootkits and the implications for the trustworthiness of any results obtained from an infected system. We acknowledge that Katana could be circumvented by attackers who manipulate the data structures, with potentially disastrous implications for the analysis results. However, we would like to point out that a DKSM attack not only has to change the desired core data structure of the operating system, but also *every single* accessor function operating on such structures to maintain system stability. This is a challenging task for an attacker, especially because updating the kernel code means that Katana will also receive the updated information from the newly generated code. The only way for an attacker to adapt would be to carefully rearrange the basic blocks of the functions inspected by our approach in a way that disassembly logic becomes oblivious to the member locations within structures. This is a non-trivial task, even for an advanced attacker.

Other approaches such as LogicMem [21] are much easier to fool. LogicMem starts its analysis by scanning the memory for the string `"swapper/0"` to find the first process in the task list. A rootkit with full access to kernel memory might create a fake task list and rename the `"swapper/0"` process afterwards. While it is possible to create a second kernel image in memory containing wrong offsets, it is not easily possible to hide the `.text` segment of the operating kernel. In such a case, Katana can detect that two conflicting kernels are placed in memory and warn the analyst.

As Katana's ideas rely solely on the structure of the `.text` segment of the kernel, the structure layout derivation is not affected by attacks like Direct Kernel Object Manipulation (DKOM). Of course, analysis passes that access manipulated objects may still be impeded by DKOM, but not to a greater extent than other tools that analyze memory.

6.3 Real World Impact

We showed that Katana can perform vital analyses of *Volatility* on most major Linux distributions, even without the presence of debugging symbols or in the presence of distributor patches. While the *Volatility* Foundation maintains a repository of pre-generated profiles for most major Linux distributions, this repository has become outdated and can, by design, not include every custom kernel built for IoT devices. If no ready-to-use profile is available, an analyst might consider switching to Katana in order to avoid the lengthy process of creating a *Volatility* profile. Moreover, those profiles can be generated only for kernels where debugging information is published, which is not the case for distributions with self-compiled kernels (e.g., Gentoo), rolling release distributions where the kernel quickly becomes outdated (e.g., Arch Linux) or IoT devices where manufactures are hesitant to provide access to build toolchains, kernel configurations, and source code. Here, a binary-only analysis is the only viable option.

Katana can also be used to generate profiles suitable for *Volatility* in order to allow reusing existing analyses. However, *Volatility*'s codebase is barely maintained and stuck on Python 2. Its Python 3 successor, *Volatility 3*, does not yet support many Linux analyses. Other binary analysis frameworks like Avatar^2 [17] could also be enriched with forensic information. Taken together, we believe Katana closes an important gap in obtaining forensic information on Linux.

7 RELATED WORK

Our closest competitor is AutoProfile [19], which evolved as concurrent research to our own. It is also based on *code-based* analysis extracting structure offsets from the kernel's code segment. While the overall derivation approach is similar, there are important differences. First, we base our analysis on Ghidra's P-Code intermediate representation whereas AutoProfile's taint engine is deeply based on x86-64 and currently only supports this architecture. The lead author stated to us on request that even though extending AutoProfile to other architectures would be possible, it would require a substantial amount of engineering effort. Furthermore, we showed Katana's capabilities to perform analysis cross-platform, i.e. to analyze a MIPS-based IoT device with a x86 profile, which is not possible at all with AutoProfile. Second, we do not rely on the repeated use of an SMT solver in the final processing step to resolve conflicting structure offsets. Repeated solving of an SMT problem containing constraints for *all* structures in the kernel will take a substantial amount of time. We assume this to be the key reason for our much quicker analysis (AutoProfile claims an analysis time of 8 hours for a 2 GB memory dump vs 20min in case of Katana).

LogicMem [21] solely analyzes the volatile *runtime data* of the Linux operating system and focuses on finding the task list inside the memory dump. This limits the number of analyses that can be performed to structures that are related to the `task_struct` structure definition. Particularly, an analysis on heap objects of the kernel is not possible. In contrast to Katana, generating the inference rules for every structure involves manual work. Furthermore, LogicMem is unable to handle structure layout randomization.

Besides these two recent proposals, a multitude of methods for monitoring the state of virtual machines (VMs) using the hypervisor have been suggested. These approaches are typically grouped into Guest Assisted, Debugger Assisted, Compiler Assisted, Binary Analysis, and Manual [2]. In the following, for every group, we highlight a few selected approaches.

Manual Approach Despite the fact that manual analysis is a tedious task, it is still chosen by a majority of academic forensic analysis projects [2]. Manual approaches rely on the human to solve the semantic gap problem and as such are naturally not scalable. *VMscope* [11] intercepts all CPU instructions executed on the VM. If an instruction performing a system call is encountered, the corresponding handler is executed. These handlers were implemented manually for every supported system call, each of which derives semantic meaning from the respective system call operation. *RAMPARSER* [10] attempts to recover key fields of a specific set of kernel structs. These key fields are recovered by reverse engineering certain manually picked functions that access or modify them. Additionally, heuristics that are specific to the respective kernel struct are employed to further deduce offsets to key struct fields. This has been shown to work well, but requires substantial manual effort for each specific struct and field that needs to be recovered. Other manually assisted approaches like *Panorama* [27] and *Ekkys* [7] rely on manual reconstruction of important kernel abstractions like processes or files.

In contrast to the previously named tools, there are also signature-based tools that perform the memory snapshot analysis in a bottom-up way, i.e., the analysis does not start from a known global pointer. Instead, a brute-force scan of all available memory is performed in order to detect interesting kernel structures. This memory scanning technique allows detection of kernel objects that have intentionally been hidden by malware, for example, by disconnecting it from the global object graph. However, all signature based detection tools generate their signatures from *known* kernel structure layouts [6, 14]. Most likely this is done to keep the false positive rate low.

Debugger Assisted Approach Besides *Volatility* and *Rekall*, which have already been discussed, also *libVMI*[12] can be put into this category with the same drawbacks. *HookMap* [26] leverages the `System.map` file to obtain the position of kernel symbols. However, the `System.map` file severely limits possibilities of analysis as no knowledge of the layout of kernel structs is contained.

Binary Analysis Approach Binary analysis approaches do not rely on any external information like debugging symbols and instead operate only on the raw binary image. *RAMAnalyzer* [28] starts out the analysis by scanning the binary image for a specific crash information string, which is generated early on in the Linux Kernel boot process. This string contains the position of certain symbols that enable recovery of the page global directory. Access to *kallsyms* then allows recovery of exported and unexported kernel symbols. To conduct analysis on the recovered symbols, structure layouts are identified in a fashion similar to *RAMPARSER* [10]. In contrast to *RAMAnalyzer*, Katana automatically recovers a wide variety of structure fields and, therefore, eases development of future analyses, which might require a completely new set of fields.

BinDiff [29] and *Diaphora* [12] are frameworks that aim to match functions between different versions of the same binary. They do not recover structure layouts by themselves. *ORIGEN* [8] uses *BinDiff*'s binary-to-binary matching in order to translate layout information between kernels. To identify structure accesses ("offset revealing instructions") in the reference kernel, they rely on a combination of static and dynamic analysis (including tracing dynamic allocations of the target types). Then, *ORIGEN* obtains a one-to-one instruction matching using *BinDiff*, and attempts to recover the offsets from the equivalent instruction in the target kernel. However, the large number of different structures makes a manual tagging approach of *all* potentially interesting types an infeasible task, and obtaining full coverage in live execution is even more difficult: E.g., consider a rare access that only happens under specific hardware conditions — ORIGEN's dynamic analysis cannot find the access and, in turn, will not be able to recover the offset. Furthermore, ORIGEN's evaluation on the kernel is limited to the `task_struct`; adding further types seems to require tedious manual work.

Compiler Assisted Approach *InSight* [24] constructs a map of kernel objects by starting with global objects and following the

[12] http://libvmi.com/

pointer members of each object. However, many code paths in the Linux kernel cast pointers to a different type or perform various other operations to them before actually accessing the pointed to memory. To avoid analysis faults caused by this dynamic behavior, *InSight* performs static code analysis on the kernel source code in order to infer the dynamic behavior of pointer members.

SigGraph [16] utilizes a custom compiler pass to infer the graph structure formed by pointers between different kernel objects of interest. By performing a brute-force scan over the entire memory, it is possible to find instances of the targeted data structures.

Guest Assisted Approach Guest-assisted approaches generally require access to the running system, such that either a program can be installed on the guest or the OS itself can be modified. Therefore, these approaches are not suitable for a post-mortem analysis setup. *Virtuoso* [5], *VMST* [9], *TZB* [4], *PoKeR* [22], and *Hybrid-Bridge* [23] try to solve the semantic gap problem by converting an in-guest analysis program to an out-of-guest analysis program, by recording one or many instruction/memory access traces of the overall system. The instruction traces are translated to an out-of-guest analysis program.

8 CONCLUSION

We presented Katana, a tool for Linux forensics. It derives symbol information of the running Linux system from the *symtab* and extends them with the symbol information made available through the *kallsyms* feature. From there, Katana is able to partially reconstruct the memory layout of central operating system structures in a *fully automated way* so that essential analyses of *Volatility* can be conducted. Our database generation is based on the concept of *accessor functions*, whose disassembly leaks information about the memory layout of kernel structs.

Furthermore, we are the first who conducted a large study on how *code-based* profile generation approaches perform for *all* structures in the Linux kernel using a set of *modern* analysis plugins on an extensive set of 85 different kernels. Another important result of our research is that forensic profiles can generalize from a default kernel configuration to other kernel configurations (like those used by most major distributions) and CPU architectures sufficiently well in order to perform common forensic analysis tasks.

We conclude that the combination of aggregate symbol information and structure layout derivation from compiled machine code is a promising approach for building robust, automated and binary-only analysis tools. Katana, which we release to the public, serves as a prototype implementation of our vision to enhance practical Linux forensics, and hopefully sparks further research ideas.

ACKNOWLEDGEMENTS

We would like to thank Fabio Pagani and Marius Muench for the inspiring discussions on our research. Furthermore, we would like to thank the numerous reviewers for their valuable feedback.

REFERENCES

[1] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 82–91. IEEE, 2010.

[2] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48(1):10, 2015.

[3] Peter Chen and Brian Noble. When virtual is better than real. In *Eighth Workshop on Hot Topics in Operating Systems*, pages 133–138. IEEE, 2001.

[4] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan Zee (north) bridge: Mining memory accesses for introspection. In *2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 839–850. ACM, 2013.

[5] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE Symposium on Security and Privacy*, pages 297–312. IEEE, 2011.

[6] 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*, pages 566–577, 2009.

[7] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Conference (USENIX ATC)*, pages 233–246. USENIX Association, 2007.

[8] 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*, pages 11–22, 2016.

[9] Yangchun Fu and Zhiqiang Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *2012 IEEE Symposium on Security and Privacy*, pages 586–600. IEEE, 2012.

[10] Richard Golden, Andrew Case, and Lodovico Marziale. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:32–40, 2010.

[11] Xuxian Jiang and Xinyuan Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *International Workshop on Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

[12] Joxean Koret. Diaphora: A free and open source program diffing tool. http://diaphora.re/, 2015-2021. Accessed: 2021-06-08.

[13] Joakim Kennedy and Avigayil Mechtinger. New Linux backdoor RedXOR likely operated by Chinese nation-state actor. https://www.intezer.com/blog/malware-analysis/new-linux-backdoor-redxor-likely-operated-by-chinese-nation-state-actor/, 2020.

[14] Bin Liang, Wei You, Wenchang Shi, and Zhaohui Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 217–227, 2011.

[15] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons, 2014.

[16] 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)*, 2011.

[17] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (Colocated NDSS Symp.)*, pages 1–11, 2018.

[18] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next generation CPU emulator framework. *BlackHat USA*, 2015.

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

[20] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *1st ACM Workshop on Virtual Machine Security (VMSec)*. ACM Press, 2009.

[21] Zhenxiao Qi, Yu Qu, and Heng Yin. LogicMem: Automatic profile generation for binary-only memory forensics via logic inference. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[22] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *4th ACM European Conference on Computer Systems*, pages 47–60. ACM, 2009.

[23] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memorization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[24] Christian Schneider, Jonas Pfoh, and Claudia Eckert. Bridging the semantic gap through static code analysis. *2012 European Workshop on Systems Security (EuroSec)*, 2012.

[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 421–432. USENIX Association, 2014.

[26] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In *International Workshop on*

Recent Advances in Intrusion Detection, pages 21–38. Springer, 2008.

[27] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer & Communications Security*, pages 116–127, 2007.

[28] Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. An adaptive approach for Linux memory analysis based on kernel code reconstruction. *EURASIP Journal on Information Security*, 2016(1):14, 2016.

[29] Zynamics. Bindiff. https://www.zynamics.com/bindiff.html, 2021. Accessed: 2021-06-08.

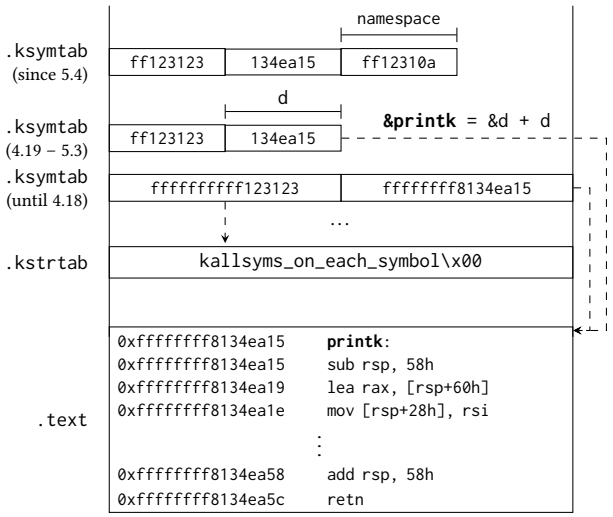

Figure 6: Structure of the symbol table on x86-64

A SYMBOL TABLE LAYOUT

The symtab is separated into two sections compiled into the kernel ELF file. The first section (`.kstrtab` or `.ksymtab_strings`) contains the ASCII representation of the *names* of all symbols separated by a zero byte (optionally compressing strings with matching prefixes). A second section (`.ksymtab`) contains pointers to the symbol names and their actual locations in memory. Figure 6 shows the structure of the symbol table across kernel versions. For space efficiency, on some 64-bit kernels starting with Linux 4.19, 8-byte absolute references inside `.ksymtab` were replaced by 4-byte *relative virtual addresses*: this encoding scheme replaces an absolute value `a` with the relative distance `d` between the target and the storage location of `d`. Since Linux 5.4, symbols are additionally organized in *namespaces* to optionally limit symbol visibility within subsystems[13]. This feature requires an additional 4-byte relative virtual address pointing to the name of the namespace to which the symbol belongs. During analysis, we scan for all possible variants.

B ANALYSIS PASSES

Table 1 lists the members of which we need to reconstruct the offsets for the first few of the analyses described in Section 5. Extracting any task-based information additionally requires the `init_task` symbol (available via the symtab), the module list requires the kallsyms-only `modules` variable (remember that data symbols are only available in presence of the `KALLSYMS_ALL` configuration option). The Linux version banner and *Dmesg* log only require global variables but no structure members.

C P-CODE LIFTING

In the following, we will give a more detailed example of how x86-64 assembly maps to *P-Code*. For this, we depicted a simple imaginary kernel function `pcode` in Figure 7 that accesses a member of its parameter and stores it in a local variable. Afterwards, the

[13] https://lkml.org/lkml/2018/7/16/566

Analysis	Data type	Member
Modules	`list_head`	`next`
	`module`	`list`
	`module`	`name`
Task listing	`cred`	`uid`
	`list_head`	`next`
	`mm_struct`	`pgd`
	`task_struct`	`comm`
	`task_struct`	`cred`
	`task_struct`	`mm` or `active_mm`
	`task_struct`	`pid`
	`task_struct`	`state`
	`task_struct`	`tasks`
Open files	`dentry`	`d_name`
	`dentry`	`d_parent`
	`fdtable`	`max_fds`
	`fdtable`	`fd`
	`file`	`f_path`
	`files_struct`	`fdt`
	`fs_struct`	`root`
	`list_head`	`next`
	`mount`	`mnt` (from Linux 3.3)
	`path`	`dentry`
	`path`	`mnt`
	`qstr`	`name`
	`task_struct`	`comm`
	`task_struct`	`files`
	`task_struct`	`fs`
	`task_struct`	`pid`
	`task_struct`	`tasks`
	`vfsmount`	`mnt_mountpoint` (from Linux 3.3)
Environment	`list_head`	`next`
	`mm_struct`	`pgd`
	`mm_struct`	`env_start`
	`mm_struct`	`env_end` (optional)
	`task_struct`	`comm`
	`task_struct`	`mm` or `active_mm`
	`task_struct`	`pid`
	`task_struct`	`tasks`

Table 1: Structure members used for the first few of the analyses described in Section 5

address of `mmap_sem` inside the local variables object is passed to the function `down_read`. When the first parameter access occurs, it is performed by a simple `mov` instruction with relative displacement to the register `rdi`.

In P-Code, values including registers and memory locations are both represented by *varnodes*: triples consisting of the relevant address space (RAM, registers, constants, and temporary values), an offset, and a size. Operations transform one or more input varnodes into an (optional) output varnode given an opcode such as INT_ADD that dictates the semantics of the operation. Here, the mapping (Kernel DB) between varnodes and x86-64 registers is

Source code:

```
int pcode(struct task_struct *t)
{
    struct mm_struct* mm = t->mm;
    down_read(&mm->mmap_sem);
    return 0;
}
```

x86-64 disassembly:

```
sub   rsp, 0x8
mov   rdi, QWORD PTR [rdi + 0x380]
add   rdi, 0x38
call  0xffffffff810b9229
xor   eax, eax
add   rsp, 0x8
ret
```

Varnode	x86-64 register
(reg, 0x20, 8)	rsp
(reg, 0x38, 8)	rdi
(reg, 0x0, 4)	rax

P-Code operations:

```
(reg, 0x20, 8)  INT_SUB (reg, 0x20, 8), (const, 0x8, 8)
(uniq, 0x30, 8) INT_ADD (reg, 0x38, 8), (const, 0x380, 8)
(uniq, 0x38, 8) LOAD (const, 0x1b1, 4), (unique, 0x30, 8)
(reg, 0x38, 8)  COPY (unique, 0x38, 8)
(reg, 0x200, 1) INT_CARRY (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x20b, 1) INT_SCARRY (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x38, 8)  INT_ADD (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x207, 1) INT_SLESS (reg, 0x38, 8), (const, 0x0, 8)
(reg, 0x206, 1) INT_EQUAL (reg, 0x38, 8), (const, 0x0, 8)
(reg, 0x20, 8)  INT_SUB (reg, 0x20, 8), (const, 0x8, 8)
(------------)  STORE (const, 0x1b1, 8), (reg, 0x20, 8),    (const, 0
    xffffffff810b7320, 8)
(------------)  CALL (ram, 0xffffffff810b9229, 8)
(reg, 0x200, 1) COPY (const, 0x0, 1)
(reg, 0x20b, 1) COPY (const, 0x0, 1)
(reg, 0x0, 4)   INT_XOR (reg, 0x0, 4), (reg, 0x0, 4)
(reg, 0x0, 8)   INT_ZEXT (reg, 0x0, 4)
(reg, 0x207, 1) INT_SLESS (reg, 0x0, 4), (const, 0x0, 4)
(reg, 0x206, 1) INT_EQUAL (reg, 0x0, 4), (const, 0x0, 4)
(reg, 0x200, 1) INT_CARRY (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x20b, 1) INT_SCARRY (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x20, 8)  INT_ADD (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x207, 1) INT_SLESS (reg, 0x20, 8), (const, 0x0, 8)
(reg, 0x206, 1) INT_EQUAL (reg, 0x20, 8), (const, 0x0, 8)
(reg, 0x288, 8) LOAD (const, 0x1b1, 8), (reg, 0x20, 8)
(reg, 0x20, 8)  INT_ADD (reg, 0x20, 8), (const, 0x8, 8)
(------------)  RETURN (reg, 0x288, 8)
```

Figure 7: Example of the mapping between *x86-64* and *P-Code* operations.

displayed in the table in Figure 7. In the corresponding *P-Code* operations, we can see that the relative displacement is carried out by an INT_ADD operation that stores the result in a temporary varnode. Then, the actual dereference happens in the LOAD operation, with the result ending up in another temporary varnode. Finally, the acquired value is copied to the varnode representing `rdi`. As a result, in order to find the member offset we are looking for, we simply need to search for an INT_ADD operation carried out on the varnode used in the LOAD operation.

Now, imagine our GCC plugin reported that the first parameter to a call to `down_read` was identified as a member access. When we encounter a CALL P-Code operation, we can map the input address back to its symbol and realize that `down_read` was called. Next, we will ask Ghidra for the varnode representing the first parameter in a function call as mandated by the calling convention. Having figured out the varnode of interest, we can now trace backwards to the first P-Code operation writing to this varnode. Quickly, we will arrive at the INT_ADD operation and can identify that the second input varnode represents the immediate offset 0x38.

D MALWARE ANALYSIS RESULTS

In this section, we will present the analysis results obtained from a RedXOR-infected machine (Subsection 5.3) in more detail, and examine how they compare to data obtained from a live sandbox[14] and manual analysis (cf. [13]).

Process memory We used KATANA to recover the userspace memory mappings of the malware process. While the human-readable part of the output (Figure 8a) only reveals the process PID, name, and UID, we can use the recovered memory layout to dump the memory contents of the process to disk. Analyzing the resulting file in a reverse engineering tool like IDA Pro or Ghidra reveals the malware's functionality. To our knowledge, the sample is not packed or obfuscated, so reverse engineering the sample directly is possible. However, in a forensic post-mortem setting, we may not have access to the sample until we extract it from the snapshot.

Open files Listing the open file descriptors (Figure 8b) reveals that the target process was started from a terminal (standard input is bound to `/dev/pts/0`), but that it redirected its output to `/dev/null`. It maintains an open reference to `/var/tmp/.2a4D53` as file descriptor 3 — which is consistent with both the automated analysis report and the manually reverse-engineered description of the backdoor's behavior.

File access history To discover other files accessed by the malware that are not currently open, we recover `dentry` objects from the allocator's memory cache (see Section 5), and create a timeline of accesses ordered by the access time (extracted from the entry's inode if it was present, and shown in Figure 8c). Besides the aforementioned temporary file, we observe the malware's accesses to its persistence mechanism in `/etc/rc*.d`, where it masquerades as a `polkit` service [13]. In entries accessed by RedXOR, the UID and GID appear corrupted; they take the correct values of 0 (for root) and 1000 (for the default user) for the other entries. To conserve space, we only show the entries related to RedXOR here, but other user activity (related to package updates and extracting the malware sample) is visible as well.

Network connections Live analysis suggests that — at least initially — only a single DNS resolution takes place. On our virtual

[14] `https://www.virustotal.com/gui/file/4f159f6a745752e3211ca1146830c86075fd8f5db60f704605a57db904dcf5c5/behavior`

```
 PID: 2859 (4f159f6a745752e ) State: 0x1 MM 0xffff92bb80f4c200 UID 0x00
Task struct @ 0xffff92bbb61c2e00 CR3 (0xffff92bb57392000         0x17392000)
```

(a) Extracted process metadata: PID, UID, name, memory mapping, and the root of the page table of the process.

```
2859 (4f159f6a745752e ) [0]: /pts/0
2859 (4f159f6a745752e ) [1]: /dev/null
2859 (4f159f6a745752e ) [2]: /dev/null
2859 (4f159f6a745752e ) [3]: /var/tmp/.2a4D53
2859 (4f159f6a745752e ) [4]: /dev/null
```

(b) Open file descriptors of the RedXOR process.

```
Timestamp                Size  Flags  UID        GID        Inode   Path
Fri Jun 04 2021 18:33:42 53901 m... 0 1314742961 2015032758 1320912 /var/tmp/.po1kitd-update-k
Sun Jun 06 2021 18:48:15 53901 .a.b 0 1314742961 2015032758 1320912 /var/tmp/.po1kitd-update-k
                            26 ma.b 0 1314742961 2015032758 263584  /etc/rc2.d/S99po1kitd-update
                            26 ma.b 0 1314742961 2015032758 263671  /etc/rc3.d/S99po1kitd-update
                            26 ma.b 0 1314742961 2015032758 263672  /etc/rc4.d/S99po1kitd-update
                             0 ma.b 0 1314742961 2015032758 525379  /var/tmp/.2a4D53
```

(c) File access history (note the corrupted UID and GID).

Figure 8: Excerpts of Katana's output analyzing a machine infected with the RedXOR malware.

machine setup, DNS lookup uses a local resolver stub, which shows up on the list of network connections (next to mDNS discovery, DHCP, and the CUPS printer server). However, the snapshot did not capture the shortlived DNS query, because the socket is closed immediately once the DNS server responds.

Katana is also able to correctly recover the process' environment variables and other system information (e.g., the ARP table), though they do not appear to contain any additional indicators of malware infection.

System or Distribution	Kernel	GCC	Kallsyms	Banner	Dmesg	Modules	Task Listing	Open Files	Environment	ARP Table	Network	File History
Ubuntu 21.04	5.11	10.3	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
Ubuntu 20.10	5.8	10.2	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
Ubuntu 20.04	5.8	9.3	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Ubuntu 19.10	5.3	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Ubuntu 19.04	5.0	8.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 18.10	4.18	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Ubuntu 18.04	4.15	7.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 17.10	4.13	7.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 17.04	4.10	6.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 16.10	4.8	6.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 16.04	4.4	5.4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 15.10	4.2	5.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 15.04	3.19	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 14.10	3.16	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 14.04	3.13	4.8	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓[p]
Ubuntu 13.10	3.11	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Debian 11	5.10	10.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Debian 10	4.19	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Debian 9	4.9	6.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Debian 8	3.16	4.9	✓[a]	✓	✓	✓	✓	✓	✓	✓	✗	✗
CentOS 8	4.18	8.3	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
CentOS 7	3.10	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Fedora 31	5.3	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Fedora 30	5.0	9.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 29	4.18	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Fedora 28	4.16	8.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 27	4.13	7.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 26	4.11	7.1	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓[p]
Fedora 25	4.8	6.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 24	4.5	6.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 23	4.2	5.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 22	4.0	5.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 21	3.17	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 20	3.11	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Fedora 19	3.9	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✗[n]

System or Distribution	Kernel	GCC	Kallsyms	Banner	Dmesg	Modules	Task Listing	Open Files	Environment	ARP Table	Network	File History
OpenSuse 15.0	4.12	7.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[n]
OpenSuse 42.1	4.1	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[n]
Arch 21-05-01	5.11	10.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Arch 20-02-01	5.5	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗[d]	✗
Arch 19-04-02	5.0	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Arch 19-02-01	4.20	8.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[p]
Android 8.1	3.18	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✗
Android 9	4.4	4.9	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗[u]	✗
Android 10	4.14	9.0[c]	✓	✓	✓	✓	✓	✓	✓	✓	✗[u]	✗
Android 11	5.4	12.0[c]	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗
Debian 10 (ARM64)	4.19	10.2.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
WR740N (MIPS32)	2.6.31	4.3	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗	✗
Tapo C200 (MIPS32el)	3.10	4.8	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗	✗

[c] Android kernel compiled using Clang instead of GCC
[a] Kallsyms is enabled, but no KALLSYMS_ALL
[6] IPv6 disabled in the target system
[d] Recovery leads to incorrect destination addresses only
[u] Analysis is unable to recover UNIX sockets
[p] Recovery cannot find SLUB's per-CPU caches of partially filled slabs
[n] Recovery fails due to a missing offset for SLAB's kmem_cache->num

Table 2: KATANA used on multiple Linux memory snapshots

```
FFFFFFFF810712D0   do_send_sig_info
FFFFFFFF810712F5   MOV        EDI, EBP
FFFFFFFF810712F7   MOV        RSI, R12
FFFFFFFF810712FA   MOV        ECX, R13D
FFFFFFFF810712FD   MOV        RDX, RBX
FFFFFFFF81071300   CALL       send_signal
FFFFFFFF81071305   MOV        RDI, qword ptr [RBX + DAT_000006d8] ❶
FFFFFFFF8107130C   MOV        RSI, qword ptr [RSP]
FFFFFFFF81071310   MOV        EBP, EAX
FFFFFFFF81071312   CALL       _raw_spin_unlock_irqrestore
```

```
FFFFFFFF81121EF0   do_send_sig_info
FFFFFFFF81121F30   MOV        RDI, qword ptr [RBX + DAT_000009f8] ❷
FFFFFFFF81121F37   MOV        RSI, R12

FFFFFFFF81121F3A   CALL       _raw_spin_unlock_irqrestore
```

Figure 9: *BinDiff* fails to map the access at ❶ to the equivalent access at ❷, instead claiming the instruction was removed.

	Ref.	1	2	3	4	5	6	7	8	9	10	R1	R2
5.8.14													
Number of members	57560	13053	15139	17831	17206	26504	37480	38341	59736	55701	40570	56651	56651
Profile coverage (%)	100.0	86.2	82.6	74.5	73.2	57.2	61.9	48.4	46.3	49.3	50.9	99.5	99.5
Correct / Wrong (%)													
– Overall	84.8 / 9.5	73.5 / 11.2	74.3 / 11.6	65.4 / 20.4	66.2 / 18.6	68.2 / 18.8	72.3 / 19.0	70.3 / 18.9	73.3 / 19.7	74.1 / 19.4	72.7 / 18.1	86.1 / 8.1	86.1 / 8.1
– Required by Volatility	80.9 / 6.4	77.3 / 8.5	78.0 / 8.5	75.9 / 9.9	70.3 / 9.4	75.2 / 7.3	78.7 / 9.2	71.5 / 8.8	77.3 / 9.2	81.6 / 7.8	76.6 / 9.2	79.0 / 9.4	81.9 / 7.2
Katana (analyses)	43	**38**	**38**	**36**	**40**	**41**	**38**	**38**	**38**	**38**	**38**	**25**	**34**
Volatility (analyses)	**52**	15	15	10	9	10	10	9	10	10	10	15	15
5.4.70													
Number of members	55783	12534	15088	18593	18986	25118	34954	28084	33007	68099	73932	55912	55912
Profile coverage (%)	100.0	99.9	92.7	90.2	84.8	82.8	61.8	66.8	65.4	50.4	45.7	99.5	99.5
Correct / Wrong (%)													
– Overall	84.9 / 9.6	74.7 / 8.4	72.8 / 11.5	77.4 / 8.5	70.2 / 15.3	79.3 / 9.4	73.3 / 16.4	72.9 / 14.3	72.5 / 16.0	71.8 / 20.8	73.6 / 17.6	86.2 / 8.0	86.1 / 8.1
– Required by Volatility	79.5 / 7.5	75.3 / 8.9	76.0 / 8.2	76.7 / 7.5	77.4 / 6.2	72.9 / 7.1	76.4 / 7.9	78.3 / 7.7	73.3 / 5.5	81.5 / 7.5	70.7 / 7.9	79.7 / 9.1	80.4 / 7.7
Katana (analyses)	42	**41**	**41**	**42**	**43**	**44**	**41**	**41**	**42**	**33**	**42**	**25**	**36**
Volatility (analyses)	**52**	15	14	15	10	9	10	10	10	10	9	15	15
4.19.150													
Number of members	53305	11907	13662	15591	19105	21505	34953	19766	20933	38619	57069	64239	64239
Profile coverage (%)	100.0	99.9	96.2	85.4	85.7	76.6	65.8	83.0	81.9	57.4	53.7	100.0	100.0
Correct / Wrong (%)													
– Overall	84.4 / 9.1	76.3 / 8.3	66.9 / 18.3	76.9 / 8.3	70.4 / 16.0	77.4 / 11.1	73.5 / 16.8	70.0 / 17.1	71.3 / 15.6	70.6 / 19.3	74.5 / 18.0	86.5 / 7.0	86.5 / 7.0
– Required by Volatility	86.3 / 4.1	83.6 / 7.5	80.8 / 10.3	79.3 / 8.1	81.5 / 6.8	87.0 / 5.5	80.8 / 10.3	78.1 / 7.3	77.4 / 7.5	76.7 / 9.6	84.2 / 8.2	73.3 / 6.2	73.3 / 6.2
Katana (analyses)	35	**35**	**28**	**41**	**29**	**41**	**29**	**30**	**33**	**29**	**30**	**34**	**34**
Volatility (analyses)	**52**	15	10	15	10	15	10	13	10	9	10	15	15
4.14.200													
Number of members	50852	11604	14345	14053	17953	14116	30092	26208	44598	32069	71410	60379	60379
Profile coverage (%)	100.0	99.9	94.2	92.8	76.5	86.7	67.5	69.3	61.6	63.0	43.2	100.0	100.0
Correct / Wrong (%)													
– Overall	85.0 / 9.8	77.2 / 8.2	78.0 / 8.7	78.1 / 8.1	77.2 / 9.1	67.8 / 17.0	74.6 / 16.0	73.5 / 17.2	75.2 / 17.8	70.0 / 19.6	72.2 / 19.7	87.1 / 7.7	87.1 / 7.7
– Required by Volatility	84.2 / 7.9	80.4 / 8.8	80.4 / 8.8	81.8 / 8.1	81.8 / 5.4	72.9 / 11.8	78.9 / 10.6	80.3 / 10.6	80.9 / 13.2	72.3 / 12.8	77.0 / 12.5	77.6 / 9.9	77.6 / 9.9
Katana (analyses)	35	**37**	**37**	**43**	**38**	**32**	**30**	**23**	**28**	**20**	**24**	**34**	**34**
Volatility (analyses)	**52**	15	15	15	15	14	9	10	10	10	10	15	15
4.9.238													
Number of members	48029	11278	11715	15081	16326	19487	26984	26324	29047	41700	42290		
Profile coverage (%)	100.0	99.9	99.3	90.1	87.7	82.4	72.8	68.9	60.8	57.5	58.0		
Correct / Wrong (%)													
– Overall	85.4 / 9.8	76.6 / 9.4	76.1 / 9.8	77.8 / 9.9	72.3 / 16.4	72.9 / 16.3	72.9 / 17.8	71.6 / 18.4	73.3 / 17.2	73.9 / 18.9	76.4 / 15.1		
– Required by Volatility	87.6 / 5.2	83.2 / 6.0	83.9 / 5.4	83.2 / 6.0	78.5 / 10.7	82.0 / 8.6	81.2 / 9.4	80.5 / 10.7	82.8 / 9.0	79.7 / 12.6	81.9 / 9.4		
Katana (analyses)	37	**36**	**36**	**36**	**28**	**28**	**28**	**24**	**33**	**19**	**27**		
Volatility (analyses)	**52**	15	15	15	10	10	10	10	14	10	10		
4.4.238													
Number of members	45633	10842	12416	12708	12091	13093	21366	24967	46264	47216	30369		
Profile coverage (%)	100.0	99.9	99.3	91.9	94.9	92.1	80.9	65.1	46.8	53.7	60.9		
Correct / Wrong (%)													
– Overall	86.0 / 9.8	77.2 / 9.9	78.2 / 11.4	77.4 / 10.1	70.6 / 16.7	71.4 / 16.6	73.4 / 17.8	73.6 / 17.8	73.6 / 18.9	76.0 / 17.6	72.0 / 19.7		
– Required by Volatility	88.4 / 4.5	84.1 / 5.3	85.4 / 5.3	85.4 / 4.6	73.8 / 9.7	75.2 / 8.3	78.8 / 9.9	77.5 / 13.2	78.6 / 11.0	77.9 / 10.3	80.7 / 12.4		
Katana (analyses)	38	**37**	**36**	**38**	**26**	**30**	**26**	**23**	**26**	**27**	**25**		
Volatility (analyses)	**52**	15	15	15	10	10	10	10	9	10	10		
3.10.108													
Number of members	39518	8285	9323	10123	13084	13953	18697	22406	28845	18267	21546		
Profile coverage (%)	100.0	99.9	93.5	89.2	87.2	78.1	69.4	69.8	59.8	62.7	53.6		
Correct / Wrong (%)													
– Overall	85.7 / 9.7	75.5 / 10.8	76.1 / 10.4	76.1 / 11.1	76.0 / 13.4	73.2 / 16.6	71.9 / 18.2	74.0 / 18.7	74.6 / 17.7	72.3 / 16.9	73.5 / 16.3		
– Required by Volatility	91.4 / 4.9	86.1 / 5.1	86.1 / 5.1	86.7 / 5.1	88.6 / 4.4	80.9 / 9.2	82.3 / 8.9	84.6 / 9.9	87.7 / 8.0	86.1 / 8.9	80.3 / 9.2		
Katana (analyses)	34	**37**	**37**	**37**	**34**	**28**	**27**	**25**	**29**	**33**	**27**		
Volatility (analyses)	**52**	15	15	15	14	10	13	10	13	10	10		

Structure layout randomization not supported

Table 3: Performance of Volatility and Katana in the presence of kernel configuration variations

RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis

Fabian Franzen
franzen@sec.in.tum.de
Technical University of Munich
Munich, Germany

Andreas Chris Wilhelmer
andreas.wilhelmer@tum.de
Technical University of Munich
Munich, Germany

Jens Grossklags
jens.grossklags@in.tum.de
Technical University of Munich
Munich, Germany

ABSTRACT

Recently proposed tools such as *LogicMem*, *Katana*, and *AutoProfile* enable a fine-grained inspection of the operating system's memory. They provide insights that were previously only available for Linux machines specifically instrumented for cooperation with virtual machine introspection frameworks. An overly controlling cloud operator can now regularly deep-inspect VMs under their control.

In this paper, we investigate how the concept of *software diversity* can be employed to remove *structural information* from the Linux kernel to *harden* it against automated analysis by the aforementioned tools. We employ a mixture of small targeted obfuscations to the memory layout and randomization of the ABI between functions in the Linux kernel as they provide predictable artifacts across different compilers, kernel configurations and the presence of *Structure Layout Randomization*.

We provide an implementation of our ideas in RandCompile, which is composed of a *small* patch set for the 5.15 Linux LTS kernel and a compiler plugin. RandCompile seeks to remove *structural information* artifacts, which we call *forensic gadgets*, to eliminate all leverage points for further analysis of the tools mentioned above. Our approach does *not* require major modifications to the kernel code base and only has a negligible performance impact (less than 5% percent), which is less than other major security or debugging features enabled by default in the Linux kernel.

CCS CONCEPTS

• **Applied computing** → **System forensics**; • **Security and privacy** → *Operating systems security*.

KEYWORDS

memory forensics, automated profile generation, binary analysis, OS obfuscation

ACM Reference Format:
Fabian Franzen, Andreas Chris Wilhelmer, and Jens Grossklags. 2023. RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3627106.3627197

ACSAC '23, December 4–8, 2023, Austin, TX, USA

ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627197

1 INTRODUCTION

Forensic analysis frameworks such as *Volatility* [12], *Rekall Forensics* [15], but also virtual machine introspection tools such as *libVMI* [30] or the operating system introspection plugins of the *Panda.re* toolkit [31] allow an analyst to observe an operating system during its execution. All aforementioned tools require a precise description of the operating system's data structures — called a profile — and a set of handcrafted analysis procedures to extract the information relevant to the analyst from the memory dump. In practice, a profile is often generated from debugging information created by the compiler. These classical approaches have high analysis accuracy, can access all information available in the memory dump, and can, therefore, offer a wide variety of introspection features to the analyst. Unfortunately, this high flexibility and accuracy come with drawbacks in practice: First, it is difficult to keep the analysis procedures up to date with operating system development [27]. Second, creating, maintaining, distributing, and identifying the appropriate profile for the operating system data structures has shown to be challenging. This is especially true for the Linux operating system as vital data structures are affected by myriads of compile time options. Furthermore, *software diversity* mechanisms such as *Structure Layout Randomization* may alter their layout.

Inferring forensic profiles used to be a major challenge if debugging information was unavailable or did not perfectly match the actual data structures. Since 2021, LogicMem [32], AutoProfile [28], and Katana [13] have been proposed to solve this problem. They derive a Linux profile solely from a memory dump at analysis time. The latter two utilize artifacts of the Application Binary Interface (ABI) between functions that is unaffected by kernel configuration and compiler choice. Furthermore, we see *OS-agnostic* forensics grow as a research field with the development of Fossil [27] and HyperLink [36]. These tools offer a less precise analysis, but still provide useful information by detecting implementation patterns shared across different OSes (e.g., the linked list of all processes).

While the advent of these tools is good news for analysts performing post-mortem forensics on systems after a security incident, it also makes it possible for malicious cloud providers to collect data on the behavior of all virtual machines under their control. Using the mentioned tools, generating process listings, inspecting the kernel log buffer, and – in the case of Katana – reconstructing the list of recently opened files of the running processes would be possible for a malicious cloud provider. Moreover, *Structure Layout Randomization*, a mainlined kernel hardening technique that interferes with common forensic tools, is not effective against many modern analysis approaches. This leaves cautious users with no defense against hypervisor attacks.

In 2022, Marth et al. [24] showed the resulting consequences by constructing a rootkit that could infect – starting from the ARM TrustZone – an unknown kernel residing in the *normal world* of the CPU. They followed the methods of LogicMem and HyperLink to allow their rootkit to function even in the presence of KASLR, *Structure Layout Randomization*, and an unknown kernel configuration.

In this paper, we propose RandCompile, a set of small compile-time modifications for the Linux 5.15 kernel that complements *Structure Layout Randomization*. By obfuscating only two fields of the central structure for managing process information and changing the ABI between functions through a compiler plugin, we will prevent analysis by forensic frameworks.

In summary, we make the following contributions:

- We collect the leverage points for analysis of different memory forensic frameworks (see Section 2). We call these leverage points *Forensic Gadgets*.
- We implement RandCompile, which consists of a plugin for the GCC compiler and a dedicated patch set for the Linux 5.15 kernel. RandCompile can be used to remove many *Forensic Gadgets* from the Linux kernel, has a negligible performance overhead (less than 5%), and can be used in addition to *Structure Layout Randomization*.
- We evaluate the effectiveness of the modifications of RandCompile against the existing forensic frameworks and rootkits and demonstrate that RandCompile provides effective protection against all of them.
- We release RandCompile[1] to the public.

2 THE STATE OF LINUX MEMORY FORENSICS

In this section, we will summarize the various techniques and frameworks commonly used to perform memory forensics on Linux.

2.1 Threat Model

All forensic frameworks, virtual machine introspection tools or rootkits discussed in the following abuse a common threat model for a normal system. In the worst case, they run with the highest privilege in the hypervisor and, therefore, can directly read from, write to, and execute from any memory location that is not protected by full memory encryption such as AMD-SEV. If tools run in a post-mortem setting, they will require at least a full memory dump including the state of all CPU registers, which reveal the position of the currently used page table. In general, we also need to assume that the adversary has full access to the compiled Linux kernel binary including its modules (i.e., by scratching it out of a memory dump, or stealing it from a disk, SD-Card or a central distribution server).

Note that a hypervisor-level malware author might not need to embed all heavyweight machinery for program analysis including tools such as the *Ghidra decompiler*, the *angr* symbolic execution engine or the *Z3 SMT solver*. Instead, we can generally assume that the machine has some sort of internet connection that allows offloading of these analyses to some remote machine. This assumption reduces implementation complexity for an analyst or attacker.

[1] https://github.com/tum-itsec/randcompile

2.2 Classical Approaches and Their Drawbacks

As previously mentioned, memory forensic tools generally require—aside from the actual memory dump—a profile that describes the data structures inside it. This can be generated from debugging information, i.e. *Volatility* and *Rekall Forensics* function in this way. Because of their *exact* knowledge of the operating system data structures, they can *potentially* extract all information that the Linux kernel has saved. This includes, for example, the list of processes currently running, the list of loaded kernel modules, the environment variables set for each process, the internal dmesg log ringbuffer of the kernel, the list of currently opened files or sockets, and the MAC addresses in the ARP cache of the system.

Unfortunately, profiles can be sensitive to even slight changes in Linux kernel versions and configuration; so central repositories with profiles can quickly become outdated [13]. Moreover, debugging information for generating a profile might be unavailable if the analysis target is an outdated Linux distribution, an embedded system, where the vendor does not release debugging information, an Android system, or a custom kernel build by an individual [32].

Last but not least, a system operator might choose to enable *Structure Layout Randomization*. Without *Structure Layout Randomization*, a compiler will place members of C-style structs in the same order as they have been declared by the programmer. However, in many cases the exact memory layout of C-structs is not relevant to the functionality of the program. *Structure Layout Randomization* allows the developer to selectively relax this constraint and will instruct the compiler to reorder the members of selected structure types. Because of this, an analyst cannot easily recreate a profile for such a kernel, *even* if he is aware of the exact kernel version and configuration. *Structure Layout Randomization* was originally meant to implement *software diversity*, which seeks to introduce uncertainty into a software system in order to prevent attacks. Such attacks can be binary exploitation attacks, side channel analysis and reverse engineering [19]. We will explain *Structure Layout Randomization* further in Appendix A.

2.3 Profile Inference Tools

If no debugging information is available, profile inference tools come into play that can derive a profile solely from the runtime data, the machine code inside the memory dump, or combinations of both approaches. We will describe important facets of the different tools in the following subsections.

2.3.1 LogicMem. To derive a profile from a memory dump, this tool constructs invariants (or rules) for interesting data structures from the Linux kernel's source code before the actual analysis. An example of such a rule is the *order rule*: The fields required for the implementation of a kernel feature are removed and added via the `ifdef` preprocessor directive. If the feature is enabled, the fields are inserted and all fields from the insertion point are shifted in memory. LogicMem can use the order, which stays constant, or match a sequence of fields not interrupted by an `ifdef` as a block.

Furthermore, the rules of LogicMem place restrictions on values a field of a specific type can get assigned. For example, pointer type

Tool	Year	Analysis Subject	FG 1: Special comm	FG 2: Symbol Tables	FG 3: ABI Constraints	FG 4: Order of Fields	FG 5: Pointer Graph	Recovery Scope
Linux-specific								
Katana [13]	2022	Offset Revealing Instructions		✗	✗			All kernel structures
Trustzone Rootkit [24]	2022	Kernel Runtime Data	✗					task_struct->{tasks, comm, pid, cred, state}
LogicMem [32]	2022	Kernel Runtime Data	✗	✗		✗	✗	Handwritten rules for selection of data structures (task_struct, mm_struct, cred, …)
AutoProfile [28]	2021	Offset Revealing Instructions		✗	✗	✗		All kernel structures
OS-agnostic								
Fossil [27]	2023	Kernel Runtime Data	✗				✗	Container data structures
HyperLink [36]	2016	Kernel Runtime Data	✗				✗	Process enumeration

Table 1: Comparison of forensic frameworks capable of analyzing the Linux kernel without debugging symbols. Focus is taken on research that has evolved since 2020.

fields need to contain a valid memory address[2] or a `NULL` pointer, while strings need to consist out of printable characters.

Unfortunately, LogicMem is not able to infer these rules automatically, so the authors of LogicMem had to infer the ruleset manually according to their algorithm. Therefore, in its current form, LogicMem can only reconstruct the layout of the `task_struct` and other structures for which the authors have created rules.

LogicMem enriches its analysis by global symbols located via the kernels' symbol table. To eliminate false candidates of this table, LogicMem searches for the well-known string `swapper/0`, which is stored inside the `comm` field of the `init_task`, and searches backwards for the ELF header of the kernel.

LogicMem's use of order rules limits the functionality of the tool when *Structure Layout Randomization* is enabled [32].

2.3.2 AutoProfile and Katana. Both tools perform code-based profile inference. The executable machine code of the operating system is itself contained inside the memory dump, which includes information about how the code accesses the data. Such sequences of instructions that can be used to recover an offset are called *offset-revealing instructions*. If *Structure Layout Randomization* is applied, the machine code is changed accordingly to address the change in the structure layout. Katana can match an *offset-revealing instruction* in machine code to the Linux source code and infer a profile. An example of how an offset-revealing instruction looks like on x86-64 architecture will be given in Section 3.4.

However, the Linux kernel is a complex and large piece of software and both analysis tools need identifiable points – matchable between source and machine code – to start the matching process between offset-revealing instructions and the type specified in the source code. The authors of Katana refer to these functions as *accessor functions* [13]. Both tools utilize the symbol table of Linux and symbol information collected by the kallsyms mechanism[3] as those reveal a majority of the addresses of the kernel's functions.

Beginning from each function, a search for offset-revealing instructions is started and structurally matched with its source code. To perform the matching, ABI constraints are used. While the software diversity introduced by compiler optimizations (such as inlining, function splitting, basic block reordering, …) makes this matching difficult, the compilation process leaves matchable patterns behind.

2.3.3 HyperLink and TrustZone Rootkit. HyperLink is a forensic tool that can enumerate the running processes without knowledge of the source code. The developers observed that many operating systems made similar design decisions for the organization of the process list.

In case of FreeBSD, Windows, Mac OS X, and Linux the data structure for a process contains a linked list of all processes and a process name stored in close, constant proximity of the pointer. For example, HyperLink scans for the string `swapper/0` on Linux. Afterward, HyperLink looks for all pointers in close proximity. Each of them is dereferenced multiple times and checked for a string that appears at a *constant offset* in all cases. If such a pointer is found, the linked list of all tasks is found.

In 2022, Marth et al. [24] combined this approach with the ideas of LogicMem. In addition to the process name and the linked list pointer, they require the location of the PID field and `cred` pointer inside the `task_struct` to elevate the privileges from TrustZone. To work in the presence of *Structure Layout Randomization*, they do not use order rules like LogicMem, but semantic rules: The PID of the first process must start at 1 and increase as the process list is traversed, which is true for the first few processes [24].

[2] LogicMem requires the page table being currently in use and can, therefore, dereference pointers to check them for validity.

[3] The `kallsym` symbol collection enables stack-traces, live-patching of specific functions and function tracing features

2.3.4 Fossil. The tool does not create a profile in the direct sense. It can use one or multiple seed values (such as names of special processes) known to an analyst to identify the process list in the overall pointer graph of the memory dump. PIDs can be identified by looking for integers in ascending order similar to Marth et al. [24]. In contrast to the previous approach, Fossil is capable of exploring the pointer structures inside operating systems in an OS-agnostic way. It has built-in support for various container data types such as doubly linked lists, arrays, and trees. Given the seed value of the first task (i.e., `swapper/0` on Linux), it can locate the containing data structure and explore other instances of this type by following the list pointers even in presence of *Structure Layout Randomization*.

2.4 Forensic Gadgets

We noticed that all forensic tools share similar leverage points for their analysis. We will refer to these leverage points as *Forensic Gadgets* and summarize our findings in Table 1.

FG 1 – Special `comm` Values. The data structure containing all relevant process information on Linux (`task_struct`) contains a fixed length 16 byte string that is embedded directly inside the structure. Multiple frameworks search for the hardcoded names of the first or the first two tasks: `swapper/0` and `init`. During our experiments, we found that the first string is only contained once in memory dumps if only kernel addresses are considered. Therefore, if such a string is found it usually directly reveals the location of one `task_struct` instance.

FG 2 – Symbol Tables. The information encoded in the Linux kernel's own symbol tables or the kallsyms mechanism – containing additional symbol information for tracing and readable stack traces – is used. These reveal the location of several global symbols (like the first task in the process list, i.e., the `init_task` symbol).

FG 3 – ABI Constraints. The System V ABI mandates the order in which *function arguments* are passed into functions. This is convenient for a profile generation tool, if the argument to a function is retrieved from a structure or used to get a member from a structure. In this case, the function can be used to match a point in the data flow graph of the assembly instructions with the data flow in the source code. The same is true for the *return values* of functions.

FG 4 – Order of Fields. LogicMem and AutoProfile make extensive use of order rules for fields within a structure. Even if the offset of a field inside a structure changes if a field is added or removed by a source code or configuration change, the order of the fields stays constant. Therefore, an identified field of a structure allows to eliminate possibilities for fields before or after the respective member. Note that this gadget is not available when *Structure Layout Randomization* is enabled.

FG 5 – Pointer Graph. An analysis tool can check if a pointer *P* is of a specific type *A* by dereferencing it and checking the memory layout constraints of type *A* on the memory location that *P* leads to. Furthermore, an analysis tool can check for specific structures of pointer chains. Consider the check for a cyclic linked list (e.g., the list of all running processes). A cyclic linked list is found if a pointer chain ends in its starting pointer and all pointers contain resolvable addresses in virtual memory or the `NULL` value.

3 RANDCOMPILE

RandCompile's modifications seek to *harden* the Linux system against *easy* access to its structural information, which is needed by analysts or attackers having the capabilities described in Section 2.1 to complete a reverse engineering or runtime information extraction task. *Hardening* means that an analyst needs to spend more computational resources or manual reverse engineering work to complete the task.

To remove the most critical sources of information for all aforementioned tools, we chose to implement three core obfuscations in RandCompile: First, by using a compiler plugin, we aim to remove the ABI patterns (FG 3) from the Linux kernel. Second, we suggest marking more structures for randomization, since the existing protection offered by *Structure Layout Randomization* already impedes the analysis of tools using the FG 4 gadget. Third, we made a few selected changes to the Linux kernel code to enable pointer encryption on core pointers (targeting FG 5) and string encryption for the `comm` field (FG 1). Table 2 summarizes which feature of RandCompile addresses a particular *forensic gadget* and which implementation method was chosen. The modifications are described in greater detail in the following subsections.

During the design of RandCompile, we preferred mitigations that can be applied to the Linux kernel in an easy manner as well as changes not degrading the performance of the running system in a notable way. Modifications such as pointer and string encryption are simple, but *force* an analyst into the more complicated inspection of the runtime code, whose analysis is now impeded by ABI randomization. We did *not* build a complete obfuscation framework for the Linux kernel, because perfect obfuscation is in general not possible [1, 14].

3.1 String and Pointer Encryption

Multiple frameworks utilize the fact that the first task in the task listing is always named `swapper/0` (FG 1). This is defined as a constant in the kernel code[4] and used throughout all papers we found using the `comm` field in their analysis. Note that the field `comm` is usually not used for listing the running processes in commands like `ps`, but predominately inside debugging messages for the Linux kernel log ring buffer `dmesg`. The process name[5], which the `ps` command displays by default, is read from the virtual memory mapping of each process on demand, along with the arguments of each process. Therefore, this name is *not* always present in the memory mapping of the kernel. Using RandCompile, we can obfuscate the string by injecting a encryption function in the `set_task_comm` function and the `get_task_comm` macro.

As part of RandCompile's kernel patches, these functions are patched to perform a simple XOR encryption with a key generated at compile time for each kernel configuration. The key is stored as a constant inside the machine code of both functions. The corresponding getter and setter functions are marked for inlining to prevent an analyst from looking them up in the symbol table (FG 2) and calling them.

[4] `include/linux/init_task.h: #define INIT_TASK_COMM "swapper"`

[5] Accessible from userspace through `/proc/<pid>/cmdline`

	Forensic Gadgets					Transformation	
	FG-1	FG-2	FG-3	FG-4	FG-5	GCC Plugin	Manual
String and Pointer Encryption	✓				✓		✓
Better Data-Order Randomization				✓			✓
Externalize `printk` Format Strings			✓			✓	
Adding Bogus Parameters with Artificial Memory Accesses			✓			✓	

Table 2: Features of RandCompile targeting different Forensic Gadgets

Pointer encryption for the fields `tasks.{next, prev}` is realized in a similar fashion. These two pointers form a circular linked list connecting all `task_structs` of the running processes. This linked list is recovered by all frameworks that utilize FG 5. By utilizing the fact that all instances of the `task_struct` share the same data layout, an analysis tool can verify its assumptions about the type of a field by iterating through the list and testing its presumed rules for a field on every instance stored inside the list. By encrypting the pointers with an XOR-key that is accessed via inlined functions by the kernel, an analysis framework cannot traverse the list in an easy manner (addressing FG 5).

3.2 Better Data-Order Randomization

The *Forensic Gadget* FG 4 is already partially addressed by the *Structure Layout Randomization* feature inside the kernel. However, the kernel developers have marked only very few structures for randomization; presumably those that have previously been used for exploiting the kernel. To optimize the protection against forensic tools, more structures should be marked for randomization. In our 5.15 kernel, we found that out of the 72 structure types that could possibly get included into the `task_struct`, only 8 have *Structure Layout Randomization* enabled. Therefore, RandCompile adds *Structure Layout Randomization* support to all types for which LogicMem's authors generated rules.

Another problem becomes apparent, if we inspect the implementation of linked lists. We can observe that they are realized by embedding a `list_head` structure containing the relevant pointers into the `task_struct`. While all fields inside the `task_struct` get randomized, the order of the fields inside the `list_head` structure stays the same. This yields small sections of data *not* being randomized.

Regarding the linked list of all tasks (field `tasks`), RandCompile embeds the list pointers *directly* inside the `task_struct`, eliminating the `list_head` structure. This allows *Structure Layout Randomization* to randomize the pointers to different locations. However, this requires specialized list traversal methods for the respective type. The special treatment of the tasks pointer is justified by the importance in the forensic work we studied (see Section 2), but it might be advisable to lift other fields of container data structures up to the `task_struct` type, too.

3.3 Externalize `printk` Format Strings

Obtaining the contents of the kernel log buffer is trivial, as the log messages are saved in readable ASCII text in a continuous memory region inside the kernel. An analyst can obtain a substantial part of its contents by looking for continuous sequences of ASCII-printable characters contained in the memory dump. A popular utility for this is `strings`.

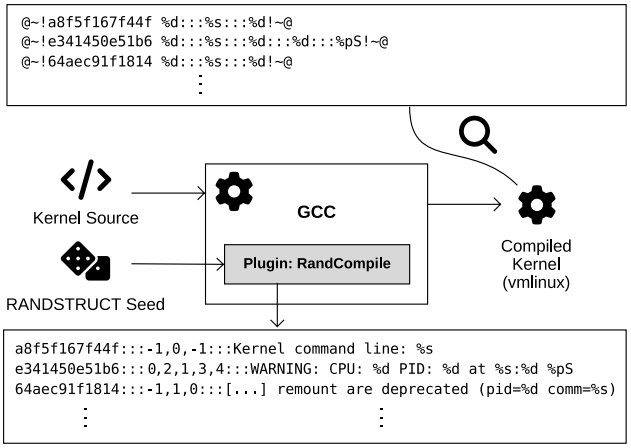

Figure 1: Overview of `printk` string externalization

Unfortunately, the kernel log buffer is also a valuable leverage point for profile generation frameworks. If the kernel reads members of structures and passes their values to the `printk` function, the function call can be used to deduce the offsets of the parameters within the respective source struct. Furthermore, the constant nature of the format string allows for the easy creation of a precise mapping of each individual instance of a function call to its source code location, even if it is called multiple times within the function under consideration and despite any potential reordering efforts.

Using logging strings or other string constants inside binary artifacts are known to be good starting points for many reversing tasks [17] and commercial obfuscation products are known to obfuscate them in special ways to prevent analysis [16]. Therefore, we conclude that this artifact needs special attention.

In order to impede the analysis of profile generation tools, our compiler plugin extracts the format strings for `printk`, if they are constant and known at compilation time, and replaces them with a unique identifier. This process is visualized in Fig. 1. During compilation with GCC, RandCompile assigns each constant format string a unique identifier. This identifier is generated by hashing the location of the format string in the source code together with a random seed value, which is created before each compilation run of the Linux kernel.

```
[    0.456668] @~!a8f5f167f44f 732845:::nokaslr:::-19872!~@
[    0.460537] @~!e341450e51b6 1234:::kernel/module.c:::1:::1108:::module_put+0x57/0x70!~@
[    0.461330] @~!64aec91f1814 948833:::sh:::1234!~@
```

dmesg | dmesgfilt mapfile

```
[    0.456668] Kernel command line: nokaslr
[    0.460537] WARNING: CPU: 0 PID: 1234 at kernel/module.c:1108 module_put+0x57/0x70
[    0.461330] [...] remount are deprectated (pid=1234, comm=sh)
```

Figure 2: Reintegration of the format strings into the log

```
unsigned long start = ..., end = ...;
mm_walk_ops prot_none_walk_ops = ...;
pgprot_t new_pgprot = vm_get_page_prot(...);
error = walk_page_range(current->mm❶, start❷,
     end❸, &prot_none_walk_ops❹, &new_pgprot❺);
```

Figure 3: Invocation of the function `walk_page_range` with *Forensic Gadgets*

In a second step, the format string is scanned for format specifiers and a replacement for the format string is generated that contains *only* the identifier and the format specifiers in randomized order. The parameters for the `printk` function are adjusted accordingly. If `printk` is invoked with less than a threshold of three parameters, RandCompile will insert additional bogus parameters from the function context. This will result in almost unreadable kernel log messages being printed out on the kernel console when being queried by the `dmesg` utility.

For the kernel console, this is usually wanted if the kernel runs in a virtualized environment. System operators usually do not want to disclose information to the hypervisor, if it is operated by a cloud operator that cannot be fully trusted. Nonetheless, benign cloud customers might want to decode the kernel log messages to a readable format. Usually, they will obtain this using the `dmesg` utility, save it and transfer it to a machine under their control. There, they can decode it using the `dmesgfilt` utility, which we provide alongside with RandCompile. Further, `dmesgfilt` works similar to the `c++filt` utility and will scan for the character sequence that marks the replacement in the dmesg output. Matching strings are looked up in the mapping file, created during compilation of the Linux kernel. The whole process is shown in Fig. 2. During this *reintegration process* previously introduced bogus parameters (marked in red) will be discarded. In the end, the user is left with a fully reconstructed `dmesg` log.

3.4 Parameter Order Randomization

In order to make matching with source code harder, our plugin will randomize the order of the parameters in as many Linux kernel functions as possible. Consider the following function call to `walk_page_range` that is visualized in Fig. 3 and its compiled version in Fig. 4.

The AMD64 Architecture Processor Supplement of the System V ABI specification mandates that function parameters are stored in the 6 registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` in this order on invocation [23]. Therefore, the first parameter (❶) is guaranteed to be passed to the function using the `rdi` register on a Linux kernel for the x86-64 platform. An automatic profile generation program

```
mov    rcx❹,0xffffffff82019c60
mov    rdx❸,r12
mov    rsi❷,rbp
mov    QWORD PTR [rsp+0x20],rax
lea    r8❺,[rsp+0x20]
mov    rax,QWORD PTR gs:0x16d00
mov    rdi❶,QWORD PTR [rax+0x440]
call   ffffffff811bacd0 <walk_page_range>
```

Figure 4: Corresponding assembly code generated by GCC for Fig. 3

can track the data flow back to the pointer dereference operation necessary to get `mm` from `current` and, therefore, determine the offset (`0x440`) of `mm` in `task_struct`.

In order to prevent such analysis, we introduce a compiler pass to shuffle the order of parameters passed to functions in the compiler's middle end. Manipulating the ABI between functions at this point allows us to leverage GCC's various architecture-independent optimization passes to optimize away potential inefficiencies and to avoid constant instruction patterns that could be easily matched by an analysis framework. As a result, a profile inference tool would now need to check **all** parameters of the **walk_page_range** function instead of only the first to recover the offset of `mm` in Fig. 4.

The main problem of this reordering approach is that parameters of a function with few parameters will be moved to the same position rather frequently. Therefore, functions with only one parameter will not be changed at all, functions with two parameters will experience a swap in parameters in 50% of the cases, and so on. To combat this, RandCompile adds bogus parameters to functions with less than 6 parameters. These additional parameters are also considered for parameter order randomization. We chose to not artificially push the number of function arguments beyond 6, since the System V call ABI designates only that many registers for efficient passing of pointer-like arguments. Further parameters would be passed to the function on the stack, which would introduce performance issues, as memory accesses are in general much slower than accessing registers. Nonetheless, RandCompile will fully randomize the order of parameters if a function should have already more than 6 parameters.

This results in any actual parameter remaining in its original spot approximately 15% ($\frac{1}{6}$) of the time (or less if the function has more than 6 parameters). An unaware analysis tool may, therefore, correctly reconstruct the offset in some cases.

3.5 Adding Bogus Parameters with Artificial Memory Accesses

Although our reordering approach outlined above raises the reordering quality for functions with a very low parameter count, not all of these parameters will be sourced from memory and assigned using offset-revealing instructions. In those cases, analysis tools expecting a parameter to be originating from a structure will not predict a wrong offset, but no offset at all. Thus, without artificially inserting offset-revealing instructions during bogus parameter generation, some fields of structs that get accessed by many different functions can still be reconstructed using statistical analysis.

To combat this, RandCompile can also fill bogus parameters with accesses to data structures already passed to a function. Consider again the example from Fig. 4. This function would receive a bogus argument in order to have 6 parameters. Instead of filling this parameter with zero, RandCompile will fill it with an equally sized value that is randomly read from a memory range residing inside of the bounds of `current`. We show the resulting AST modification of RandCompile (translated back to C-syntax) in Fig. 5 and the matching compiled down version in Fig. 6.

The offset for new parameters will not be chosen completely randomly. This would result in plenty of potential random offset candidates and a set of correct offsets if the parameters have been randomized to their original position. Therefore, tools such as Katana – using a majority voting approach [13] in the last step of determining the final offset – could still find the correct one, as the random ones are of low chance to occur more than once. Thus truly random offsets simply lead to a vast increase in random noise, whereas the remaining 15% of cases in which the actual parameter can be found in its original register uphold the pattern exploited by Katana in the first place.

Instead, we added a pseudorandom offset selection to our bogus argument generation. This generates a set of n plausible alternative offsets for a particular field in a deterministic way, which will give bogus parameters a reasonable chance to be considered plausible in the majority voting of Katana, while still being random enough to hide the correct value. Note that the matching process of analysis tools is not perfect. Empirically, we found the value $n = 3$ to be best suited for our needs. We describe our experiments more closely in Appendix B; Figure 8 visualizes the process.

3.6 Practical Implementation Aspects

We chose to implement our approach as a middle-end GCC plugin, allowing for a combination with the existing structure layout randomization plugin of the Linux kernel. Furthermore, middle-end plugins are architecture-agnostic and have access to almost all information of the original AST created by the C-parser, which is present as a GIMPLE tree. In the middle-end phase, the GCC compiler has not yet performed any register allocation or most of its optimizations.

```
unsigned long start = ..., end = ...;
mm_walk_ops prot_none_walk_ops = ...;
pgprot_t new_pgprot = vm_get_page_prot(...);
struct mm_struct *tmp = (char*)current)+0x10;
error = walk_page_range(&prot_none_walk_ops❹,
     end❸, start❷, current->mm❶, tmp❻,
     &new_pgprot❺);
```

Figure 5: The function `walk_page_range` randomized with RandCompile

```
mov     rdi❹,0xffffffff82019c60
mov     rsi❸,r12
mov     rdx❷,rbp
mov     QWORD PTR [rsp+0x20],rax
lea     r9❺,[rsp+0x20]
mov     rax,QWORD PTR gs:0x16d00
mov     r8❻,QWORD PTR [rax+0x10]
mov     rcx❶,QWORD PTR [rax+0x440]
call    ffffffff811bacd0 <walk_page_range>
```

Figure 6: Corresponding assembly code generated by GCC and RandCompile for Fig. 5

4 EVALUATION

First, we evaluate the effect of RandCompile on all the tools collected in Section 2. We pay special attention to Katana and the HyperLink/TrustZone rootkit as their power is not affected by the existing *Structure Layout Randomization* feature of the Linux kernel. In the case of Katana, we did a thorough evaluation, including tests with its available implementation on different versions of our 5.15 kernel. First, we test the kernel with RandCompile disabled, then with variants of RandCompile's hardening configurations. Subsequently, we evaluate how each of RandCompile's obfuscation mechanisms contributes to its overall protection.

For our comparison with the HyperLink/TrustZone rootkit, we reimplemented HyperLink as a plugin for GDB and integrated additional analyses from the TrustZone rootkit. These additional analyses reveal, for example, the PID in addition to solely the process name. The evaluation of RandCompile against Fossil is done on a theoretical basis as Fossil requires an analyst to manually interpret the data. Finally, we check if RandCompile impedes the overall performance of a system by using the microbenchmark *lmbench3*.

4.1 Impact on Forensic Frameworks

4.1.1 HyperLink and TrustZone Rootkit. We will evaluate HyperLink and the TrustZone rootkit together, because both tools share the same implementation ideas. To provide the rootkit functionality, the rootkit implemented recovery of additional layout information and we will, therefore, focus our analysis on it (see Section 2.3.3).

We reimplemented the rootkit as a Python plugin for GDB, where it can be used in conjunction with QEMU to perform the first three analysis steps: 1) the recovery of the process list in memory; 2) the identification of the `tasks.next` pointer so that the rootkit can find the userspace process of the attacker process and elevate its privileges; 3) the identification of the `comm` and `pid` fields, so that the target process can be identified inside the list. The missing final step eventually uses the knowledge gathered by the first steps to perform the elevation of the privileges to root. In the original implementation, the rootkit looked for a field with the same offset on all tasks in the process list with ascending PIDs without gaps. We modified this rule. On our 5.15 kernel for x86-64, we just require an integer field with ascending numbers and removed the *no gaps* constraint as there are gaps, even in the processes started early at boot on our system.

We found that using RandCompile does not allow the rootkit to find the initial location of the `init_task` symbol. This is because the rootkit uses the hardcoded string "swapper/0" to find it. Instead of two occurrences in memory without RandCompile (one in the physmap[6], one in the normal virtual memory in the kernel), our version of the rootkit now finds four occurrences. Again, these are

[6]In addition to the virtual memory mappings, Linux maps all its physical memory continuously to one point in virtual memory.

effectively two occurrences because each search hit has a corresponding hit in the physmap of the kernel. However, these two hits are not located inside the `init_task` symbol, but on the kernel stack. We consider these to be leftovers from printk statements during kernel boot, where the `comm` string was accessed to print a log message. We inspected the search hits on the kernel stack, but could not find a pointer nearby that referred to the actual `init_task`.

Furthermore, the pointer encryption will not allow our rootkit implementation to find the next and prev pointer even when we aid the tool by artificially feeding it with the location of the `init_task` as a starting point.

Eventually, the rootkit verifies that the correct `tasks.next` pointer has been found by checking whether the `prev` pointer leads to the pointer from which we originated. The `prev` pointer is found by a constraint induced by the `list_head` structure containing both pointers in pairs. This structure is not randomized and forces the compiler to put the `prev` pointer directly after the `next` pointer. However, inside a RandCompile kernel these two pointers have been moved into the scope of the surrounding `task_struct`, so that this structural constraint does not hold anymore.

4.1.2 Katana and AutoProfile. An effect on Katana is already noticeable in the first analysis step, i.e., the collection of the kernel symbols. Katana collects these by *calling* a function to iterate over all symbols saved in the kallsyms database in an emulated environment representing the execution on the analysis machine. As Katana assumes default calling conventions, the registers are not correctly set up and no symbols are collected. For further analysis, we will patch Katana to parse symbol information without calling kernel functions. This gives us more insights into the effectiveness of each transformation step of RandCompile.

For this, we disable RandCompile completely, set the 5.15 kernel to its default configuration with *Structure Layout Randomization* enabled, and initiate Katana to create a profile for forensic analysis. Next, we check to see if Katana's analysis plugins are still working. For each analysis, we check which part of the profile was needed for the plugin to work, and how many fields within that profile were inferred correctly. We selected a subset of four of Katana's analysis plugins for this practical evaluation and refer to it as *Base* state. The selected analyses are: 1) listing the modules currently loaded into the Linux kernel, 2) listing the running processes, 3) listing the opened files of the running processes, and 4) extracting the kernel dmesg log ring buffer. Each analysis needs a certain set of layout information in order to function and the analysis will fail if even *a single member* could *not* be restored. Thus, only one incorrect offset (e.g., a missing tasks pointer) would, for example, cause the whole task listing analysis to fail.

As can be seen in Table 3, Katana is able to perform all analyses on a pure *Base* kernel (i.e., not modified by RandCompile). If we redo the analysis with a fresh profile for a kernel with RandCompile fully enabled, only the listing of currently loaded kernel modules analysis is still working.

Table 3 also shows the analysis results of Katana for different weakened variants of RandCompile. *RandCompile (no bogus)* refers to a kernel with a naive reordering of arguments as described in Section 3.4, *RandCompile (-printk, -memref)* and *RandCompile (-printk)* refer to kernels with bogus arguments added to function calls (see Section 3.5), without and with the dmesg log obfuscation of Section 3.3.

If only the randomization of argument order (*RandCompile (no bogus)*) is applied, the recovery rate of structure members drops for the task listing and list files analysis. This is in line with our expectations: Both analyses are more complex and require more recovered offset information. Nonetheless, the drop is small. For both analyses only one member cannot be reconstructed. If we enable the addition of arguments filled with zero values (*RandCompile (-printk, -memref)*), the discovery rate of Katana drops drastically for the list of open files analysis.

Adding randomization, especially to `printk`, which must also be applied to the format string (see Section 3.3), has no visible effect on the recovery rate for all analyses but the dmesg log. However, the number of votes for members is reduced, making it less likely that the correct offset can be reconstructed (see Table 4 in the Appendix for more detailed numbers).

Taking a more detailed look at the number of votes cast for each member by Katana in Table 4, we can see that RandCompile has the most impact on those members that are encountered and reconstructed via a variety of different function calls.

In particular, this applies to `mm`, for which initially more than 100 occurrences to vote on had been found. Adding and shuffling artificial `NULL` parameters was able to reduce this number to merely 15 occurrences. Since no alternative offsets were presented by this approach, however, a significant bias towards the correct offset persists. If we replace the artificial `NULL` parameters with actual values sourced from memory instead, we can see that the total number of occurrences rises again but the bias shifts towards a wrong offset, causing Katana to select the wrong offset.

All our insights should also apply to AutoProfile, which we only evaluate conceptually. AutoProfile is implemented similarly as Katana with a few differences. The authors decided to use the *angr* framework instead of *Ghidra*. Furthermore, rather than using majority voting, Z3 is used as a constraint solver. The more artificially introduced or otherwise wrongly deduced offsets exist, the more complex it becomes to create a valid matching. Looking

	Base	RandCompile (no bogus)	RandCompile (-printk, -memref)	RandCompile (-printk)	RandCompile (full)
List Modules	✓	✓	✓	✓	✓
Members reconstructed	2	2	2	2	2
Task Listing	✓	✗	✗	✗	✗
Members reconstructed	6	5	5	4	4
List Files	✓	✗	✗	✗	✗
Members reconstructed	16	15	8	7	7
Dmesg Log	✓	✓	✓	✓	✗

Table 3: Evaluation of the effectiveness of RandCompile's different code transformations against Katana

at `mm`, we can also see a clear increase in available options for this heuristic, potentially defeating AutoProfile's analysis attempts.

We can leverage the fact that our plugin can reliably prevent reconstruction of more commonly used members and that partial reconstruction of kernel data structures still prevents all but one of the tested analysis procedures.

4.1.3 LogicMem. Unfortunately, we could not run LogicMem on our memory dumps as the authors did not document the input format required and we have not been able to deduce it from their code in reasonable time. Instead, we did an inspection of the ruleset the authors derived from the Linux kernel and LogicMem's source code. The LogicMem authors decided to move away from their original Prolog implementation that considered all facts available to a new Python implementation for performance reasons [3]. In contrast to the original, this implementation does not fully utilize their fact database. For example, LogicMem relies heavily upon the presence of the now obfuscated `comm` string and the encrypted `tasks` list pointers. They only consider the constraint of a constant offset between the `comm` and `tasks.next` field for the detection of the latter (like HyperLink). The presence of a detectable `comm` and the `tasks` field is, therefore, a hard requirement for LogicMem to function. As this requirement is programmed in and not expressed through rules we expect that LogicMem can also not be *easily* repaired.

Last but foremost, many of the LogicMem rules rely on order constraints that are invalidated in the presence of *Structure Layout Randomization*. As we have *extended* the usage of *Structure Layout Randomization* to several structures that have not been protected before, LogicMem's recovery rules for additional structures will also fail (e.g., `files` or `fdtable`). While LogicMem was already not able to recover a working profile for the `task_struct` with *Structure Layout Randomization*, the amount of information it could use now is even further reduced.

4.1.4 Fossil. To detect virtual addresses, *Fossil* requires a Boolean oracle function (called Ω), which the analyst needs to provide. It should be capable of deciding whether an offset in the memory dump is a valid pointer. When analyzing a RandCompile kernel, this function cannot be provided for our encrypted pointers without first reverse engineering the compiled kernel. Because of that, *Fossil* will be unable to detect the doubly linked list of all processes. However, as *Fossil* can make sense of other container structures, it might detect subsets of processes that are connected via unencrypted pointers.

Furthermore, the encryption of the `comm` string will not allow an analyst to use this as a seed to obtain one of the subsets mentioned above. Besides `comm`, we are not aware of any other easily obtainable string seed an analyst can use. It is the only string that is directly included inside the `task_struct`. Note that the `cmdline` string does *not* reside in kernel space memory, but in the memory of each userspace process. While we have not evaluated the effects of RandCompile on *Fossil*, we conclude that easy leverage points for analysis are removed here, too.

4.2 Runtime Performance Impact

We deployed our test kernels in a VM with 1 GB of RAM on a machine with an AMD Ryzen Threadripper 2950X processor we had exclusive access to. We tested RandCompile's performance using *lmbench3*[7]'s latency benchmarks and show the results in Fig. 7. All measured latencies have been normalized to one for better readability, as the respective system calls vary in speed by several orders of magnitude. The baseline measurement has been generated with an identical kernel configuration including active *Structure Layout Randomization* but with all of RandCompile mitigations disabled. Further, we added error bars indicating the standard deviation.

In general, RandCompile (with all transformations enabled; depicted as *Full*) adds less than 1-3 percent overhead to the system. In the worst case, it degrades system performance by no more than 6 percent (*open/close* benchmark). In the best case, no overhead is measurable. For comparison, the runtime overhead of the kernel's own *Ftrace* feature, which is enabled by default on almost all Linux distributions, is 4 percent for the *open/close* system call benchmark. Like enabling RandCompile, enabling *Ftrace* causes every function in the kernel to be modified by the compiler. However, here the compiler adds a patch point filled with NOP instructions until the user enables *Ftrace* at run time. Note that the 4 percent decrease in performance is measured with *Ftrace disabled* by the user.

Predominantly, the performance overhead seems to be caused by the introduction of additional bogus arguments (see Section 3.4). Pointer and string encryption (*Only Obf.*) as well as the shuffling of their parameters (*NoBogus*) do not add performance overhead that cannot be explained by noise.

The slowdown caused by adding bogus arguments to functions with less than six arguments (see Section 3.5; depicted as *NoMemRef*) was unexpected. While our change is not equivalent to changing the order of the registers in the ABI, we assumed that setting CPU registers to zero is a neglectable operation on modern CPUs. Nonetheless, the performance of *stat* is degraded by up to 8.53% (*open/close*). We assume that our implementation interferes with various optimizations of the GCC compiler (i.e., the *constprop* optimization creating specialized functions with our bogus parameters removed) running after RandCompile's compilation pass.

5 RELATED WORK

Approaches like our parameter-order randomization (Section 3.4) are also implemented in obfuscators like *Tigress* [7]. However, *Tigress* reports issues with variadic args and indirect function calls, which RandCompile can handle to the necessary extent when compiling the Linux kernel.

Instead of obfuscating the memory image, an operating system might also limit itself to detecting if it is currently being analyzed by a hypervisor [34]. However, in a forensic setting, this is not practicable as the virtual machine cannot alter its state once the memory dump has been taken.

An approach that also uses GCC to implement a variety of data structure randomization techniques is presented by Stanley et al. [33]. Their approach includes layout randomization for data structures and stack layout reordering for function parameters, to

[7] https://github.com/intel/lmbench

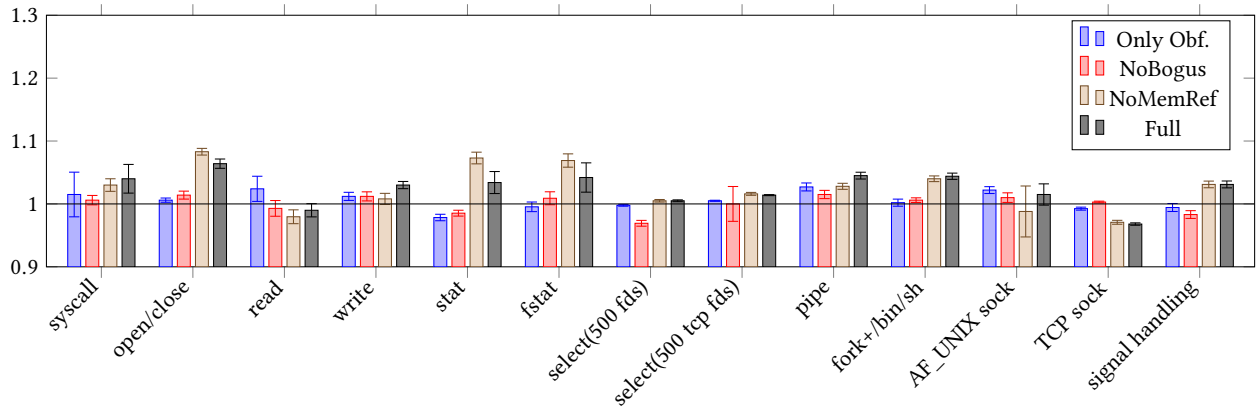

Figure 7: Latency tests of RandCompile using *lmbench3*. (lower is better, all latencies are shown normalized to 1)

counter large-scale binary exploitation attacks by introducing software diversity. It is similar to our approach in that it also performs some sort of randomization that affects the order of function parameters. However, it does work on a different abstraction level, does not randomize functions containing *varargs* at all and unconditionally excludes functions with indirect calls.

Of course, a function's calling convention is only a small part of any program's ABI. Some other approaches, which are in part architecture-specific or language-specific, target other parts of binaries' ABIs to harden them against binary exploitation and reverse engineering. One such approach targeting the Itanium C++ ABI standard's handling of vTables, which are used to model polymorphism in C++ programs, has been presented by Demicco et al. [10]. Junod et al. [18] present a language-independent, LLVM-based code obfuscation tool. The tool changes a program's control flow by merging functions and inserting additional instructions. But, besides simple reordering approaches, some techniques presented have a measurable effect on performance and code size.

Crane et al. [9] and Larsen et al. [19] present a discussion of the current state and limitations of code randomization approaches as well as software diversity in general. Furthermore, they discuss the reasons for the lack of widespread adoption of existing techniques other than address space- and structure layout randomization.

Almost all of the approaches mentioned above have the primary goal of hardening software against binary exploitation, reverse engineering and code reuse attacks. Since our plugin is primarily intended to be used as a supplement to *Structure Layout Randomization* on the Linux kernel, we chose to use the same compile-time randomization technique as the Linux kernel's implementation of this technique. Related approaches are described by Chen et al. [6] and Lin et al. [22]. However, Lin et al. [22] suggest randomizing projects differently if recompiled at different times, which would prevent plugins like kernel modules in the context of Linux.

There is also an academic implementation of dynamic structure layout randomization [5], which periodically re-randomizes data structures at runtime. Unlike classical static approaches, this technique introduces a high average performance overhead of 15% and, to our knowledge, has not been widely adopted.

6 DISCUSSION

In the following section, we will discuss the potential impact of our research and name open issues.

Correctness. To the best of our knowledge, the modifications of RandCompile will not impede the normal functioning nor the security of the overall system. Like other compiler optimizations, ABI randomization and pointer encryption are semantically sound transformations of well-defined programs. For example, the ABI is not consistent across architectures and not mandated by the C-standard. However, it is not uncommon in the Linux kernel to make assumptions about the layout of the generated code and the compiler optimization being performed. Certain functions are implemented in pure assembly and expect standard calling conventions. Similarly, functions implemented in C can be called from inline assembly sections or assembly sections. RandCompile handles these cases by scanning for such functions beforehand and does not consider them for ABI-randomization through the use of a blacklist. Likewise, string and pointer encryption do not change the semantics of a program. If we encrypt a value on store and decrypt it on load this operation is transparent to the underlying program.

Security Considerations. The use of pointer and string encryption raises the question of where to store the key securely inside the kernel. RandCompile stores the encryption key of a structure field as an immediate value of a machine code instruction inside the code. A simple analysis for high-entropy values that would unveil it in main memory is usually not efficient there. At least on x86-64 and ARM, machine code is stored in variable-length encoding that has a high entropy. This forces an analyst to perform an analysis of the Linux kernel text segment, but the matching of the corresponding encryption key for the field requires the tools to solve the problem faced by Katana and AutoProfile. These tools are impeded in their usability by the usage of ABI randomization.

A further concern might be if the encryption of values might create new *forensic gadgets* revealing the position of interesting objects in memory. Kernel pointers are easily identifiable because they are usually 8-byte aligned, have a canonical form on x86-64 and ARM and yield a valid memory location if dereferenced. An

encrypted value cannot get dereferenced without knowledge of the key and, therefore, cannot reveal knowledge about the location of other objects in memory.

Moreover, the initial search for the string *swapper/0*, which is padded with zeros to a fixed 16-byte value, will hit by chance with a significantly lesser probability than a search for a 16-byte high-entropy value. Through the introduction of string encryption, we increased the number of false positives that an analyst will face.

Binary Exploitation Defense. Randomizing the ABI of function calls inside the kernel makes its exploitation harder, because kernel exploits frequently call kernel functions [25]. In 2022, an exploit for the Linux kernel was presented [29] that calls the `set_memory_x` function in order to elevate its privileges. If such functions are randomized, the attacker needs to guess the right calling convention for this function. Furthermore, the encryption of pointers is a common exploit mitigation strategy to prevent an attacker from modifying vital pointers in the GNU libc [21] and on Windows. While Kernel Address Space Layout Randomization (KASLR) already protects the kernel by randomizing the load address of the kernel on each startup and *Stack Canaries* usually protect the return address from overflows on the kernel stack, stack-based and heap-based buffer overflows can still be exploited in the kernel code. Structure Layout Randomization was originally also designed to combat exploitation through the software diversity concept. This is especially interesting in remote exploitation scenarios or if the local administrator does not give read rights to the `/boot` folder with the Linux kernel.

Full Memory Encryption Using AMD-SEV. Given the availability of confidential computing technologies such as AMD-SEV, one may debate whether our approach is needed to combat forensic analysis. AMD-SEV provides virtual machines with the ability to transparently encrypt their memory, making it completely inaccessible to other virtual machines and even the hypervisor.

However, AMD-SEV is only available on AMD CPUs, not offered by every cloud provider, and has been subject to several bugs [11, 20, 26, 35] since its introduction. In 2021, Buhren et al. presented a hardware architectural attack on AMD-SEV that allows an attacker with physical access to obtain the encryption keys for the main memory and a fully decrypted dump of its memory contents [4]. Attacks that exploit bugs in hardware drivers to generate a memory decryption oracle despite the presence of AMD-SEV are another prominent problem [20]. Considering these attacks, we believe RandCompile can meaningfully *complement* AMD-SEV as a defense-in-depth measure by cloud customers or stand on its own in cases where AMD-SEV is not available.

Kernel Modules. RandCompile works across the boundaries of C translation units. All modifications to the ABI and the keys used for obfuscation are derived in a deterministic way from a static seed. This approach is also taken by the *Structure Layout Randomization* feature. Therefore, Linux kernel modules compiled with RandCompile support kernel modules precisely under the same conditions as a kernel compiled with *Structure Layout Randomization*, i.e. the utilized randomization seed must be available when the module is compiled.

Influence on Debugging. As RandCompile's purpose is to impede the reasoning about the kernel structure for analysis and reverse-engineering tools, this feature will make a Linux kernel *less* debuggable. Because of that, the Linux kernel should be compiled without RandCompile when conducting active development on the kernel. However, this does not affect the usefulness of RandCompile, since other security features like KASLR and *Structure Layout Randomization* will be disabled likewise in this case. Debugging information can be modified and stored separately from the executable kernel such that debugging is still possible if a critical bug should occur on a production system.

Limitations and Future Work. While already sufficient to address attack approaches of state-of-the-art tools, RandCompile could be more usable if all transformations would be applied in an automated way. This would allow its application on a wide range of Linux kernels without the need for large-scale source code changes. Pointer and string encryption are currently only implemented for a few fields in a manual way. This is because, an alias and flow analysis is needed to identify all locations that read or write to a specific location in memory to make sure that encryption and decryption are performed consistently. However, we believe that there is still room for further compiler assistance to perform more of these transformations. This would allow to encrypt more strings and pointers (on non-performance-critical paths) to further increase the resilience against high-entropy attacks trying to sidestep RandCompile. This could also permit to include protections against *FG 2 Symbol Tables*, which are currently not included in RandCompile.

Second, we would like to more closely investigate the blacklisting feature to support the use of precompiled modules and drivers for the Linux kernel as they appear in the Android ecosystem. Through the blacklisting of all imported functions of kernel modules, these modules should function without changes. This support should already be there today. However, further research is needed to evaluate if RandCompile's protection is still sufficient in these cases. Presumably, drivers use only a small surface of the Linux kernel API and structures, which would not affect the protection at all, but this is unknown so far.

7 CONCLUSION

We have presented RandCompile, a tool to remove *Forensic Gadgets* from the Linux kernel. Without these gadgets, all forensic tools we are aware of are impeded in their analysis capabilities. Primary use cases for RandCompile are the hardening of a system against automated forensic analysis by a malicious cloud provider, rootkit attacks and reverse engineering (see Section 2.1). Likewise, it could be leveraged by distributors of Android or IoT devices to hinder reverse engineering of proprietary kernel modules on their devices. Furthermore, like *Structure Layout Randomization*, which is already deployed inside the Linux kernel, it can also harden the system against binary attacks if the Linux kernel's binary is unknown to an attacker. This could be the case if the system is attacked via a network connection. We have also argued that RandCompile can be used in addition to AMD-SEV to offer a system that is more fault-tolerant to future attacks.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback.

REFERENCES

[1] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im)possibility of obfuscating programs. In *Advances in Cryptology — CRYPTO 2001*, Joe Kilian (Ed.). Springer Berlin Heidelberg, 1–18. https://doi.org/10.1007/3-540-44647-8_1

[2] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Diego Zamboni (Ed.). Springer Berlin Heidelberg, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[3] bitsecurerlab. 2022. LogicMem - Github Repository. https://github.com/bitsecurerlab/LogicMem. (Online; accessed 25-May-2023).

[4] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2875–2889. https://doi.org/10.1145/3460120.3484779

[5] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. 2015. A Practical Approach for Adaptive Data Structure Layout Randomization. In *Computer Security – ESORICS 2015*, Günther Pernul, Peter Y. Ryan, and Edgar Weippl (Eds.). Springer International Publishing, Cham, 69–89.

[6] Zhongtian Chen and Hao Han. 2017. Attack Mitigation by Data Structure Randomization. In *Foundations and Practice of Security*, Frédéric Cuppens, Lingyu Wang, Nora Cuppens-Boulahia, Nadia Tawbi, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 85–93.

[7] Christian Collberg. 2023. The Tigress C Obfuscator. https://tigress.wtf/index.html. (Online; accessed 30-September-2023).

[8] Manuel Costa, Jean-Philippe Martin, and Miguel Castro. 2008. *Data Randomization.* Technical Report MSR-TR-2008-120. 14 pages. https://www.microsoft.com/en-us/research/publication/data-randomization/

[9] Stephen Crane, Andrei Homescu, and Per Larsen. 2016. Code Randomization: Haven't We Solved This Problem Yet?. In *2016 IEEE Cybersecurity Development (SecDev)*. 124–129. https://doi.org/10.1109/SecDev.2016.036

[10] David Demicco, Rukayat Erinfolami, and Aravind Prakash. 2021. Program Obfuscation via ABI Debiasing. In *Annual Computer Security Applications Conference (ACSAC)*. Association for Computing Machinery, 146–157. https://doi.org/10.1145/3485832.3488017

[11] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. 2017. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090* (2017).

[12] Volatility Foundation. 2023. Volatility Framework. https://www.volatilityfoundation.org/. (Online; accessed 25-May-2023).

[13] Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Association for Computing Machinery, 214–231. https://doi.org/10.1145/3545948.3545980

[14] Shafi Goldwasser and Guy N. Rothblum. 2007. On best-possible obfuscation. In *Theory of Cryptography*, Salil P. Vadhan (Ed.). Springer Berlin Heidelberg, 194–213.

[15] Google. 2023. Rekall Forensics. https://github.com/google/rekall. (Online; accessed 25-May-2023).

[16] Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. 2018. Honey, I shrunk your app security: The state of Android app hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 69–91.

[17] Muhui Jiang, Lin Ma, Yajin Zhou, Qiang Liu, Cen Zhang, Zhi Wang, Xiapu Luo, Lei Wu, and Kui Ren. 2021. ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 734–748. https://doi.org/10.1145/3460120.3484753

[18] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. 3–9. https://doi.org/10.1109/SPRO.2015.10

[19] Per Larsen, Stefan Brunthaler, and Michael Franz. 2014. Security through diversity: Are we there yet? *IEEE Security & Privacy* 12, 2 (2014), 28–35. https://doi.org/10.1109/MSP.2013.129

[20] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security)*. USENIX Association, 1257–1272. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[21] GNU Libc. 2013. Pointer Encryption. https://sourceware.org/glibc/wiki/PointerEncryption.

[22] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. 2009. Polymorphing Software by Randomizing Data Structure Layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Ulrich Flegel and Danilo Bruschi (Eds.). Springer Berlin Heidelberg, 107–126.

[23] H. J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2022. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0. https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

[24] Daniel Marth, Clemens Hlauschek, Christian Schanes, and Thomas Grechenig. 2022. Abusing Trust: Mobile Kernel Subversion via TrustZone Rootkits. *2022 IEEE Security and Privacy Workshops (SPW)* (2022), 265–276.

[25] Keegan McAllister. 2012. *Writing kernel exploits.* Presentation Slides. Georgia Institute of Technology. https://tc.gtisc.gatech.edu/bss/2014/r/kernel-exploits.pdf

[26] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. *2021 IEEE Security and Privacy Workshops (SPW)* (2021), 444–455.

[27] Andrea Oliveri, Matteo Dell'Amico, and Davide Balzarotti. 2023. An OS-agnostic Approach to Memory Forensics. *2023 Network and Distributed System Security Symposium (NDSS)* (2023), 16 pages. https://doi.org/10.14722/ndss.2023.23398

[28] Fabio Pagani and Davide Balzarotti. 2021. AutoProfile: Towards Automated Profile Generation for Memory Analysis. *ACM Transactions on Privacy and Security* 25, 1, Article 6 (2021), 26 pages. https://doi.org/10.1145/3485471

[29] Samuel Page. 2022. Writing a Linux Kernel Remote in 2022. https://blog.immunityinc.com/p/writing-a-linux-kernel-remote-in-2022/. (Online; accessed 30-September-2023).

[30] LibVMI Project. 2015. LibVMI. https://libvmi.com/. (Online; accessed 25-May-2023).

[31] Panda.re Project. 2023. PANDA's OS-specific introspection plugins. https://github.com/panda-re/panda/tree/dev/panda/plugins/osi. (Online; accessed 25-May-2023).

[32] Zhenxiao Qi, Yu Qu, and Heng Yin. 2022. LogicMEM: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. *2022 Network and Distributed System Security Symposium (NDSS)* (2022), 17 pages. https://doi.org/10.14722/ndss.2022.24324

[33] Dannie M. Stanley, Dongyan Xu, and Eugene H. Spafford. 2013. Improved kernel security through memory layout randomization. *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)* (2013), 10 pages. https://doi.org/10.1109/PCCC.2013.6742768

[34] Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K. Lengyel, and Kyle J. Temkin. 2018. Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. *Digital Investigation* 26 (2018), S98–S106.

[35] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. 2021. undeSErVed trust: Exploiting permutation-agnostic remote attestation. In *2021 IEEE Security and Privacy Workshops (SPW)*. 456–466.

[36] Jidong Xiao, Lei Lu, Haining Wang, and Xiaoyun Zhu. 2016. HyperLink: Virtual machine introspection and memory forensic analysis without kernel source code. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. 127–136.

A STRUCTURE LAYOUT RANDOMIZATION

Structure Layout Randomization is a compiler plugin that has been introduced into the Linux kernel source tree back in 2017 with release 4.13. The compiler plugin was originally developed as part of the GRSecurity project, but was later mainlined into the Linux kernel. Its purpose is to harden Linux during compilation against memory corruption attacks. Without *Structure Layout Randomization*, a compiler will place members of C-style structs in the same order as they have been declared by the programmer. However, in many cases the exact memory layout of C-structs is not relevant to the functionality of the program. *Structure Layout Randomization* allows the developer to selectively relax this constraint and will allow the compiler to reorder the members of specially marked structure types. ABI compatibility through different translation units is guaranteed by basing the shuffling process on a seed shared between the different translation units.

This concept has been discussed in academia since 2008 [2, 8] with the original goal to harden the software against binary exploitation attempts if attackers are unable to access the executable binary. This is the case, if an attack is performed remotely or if the attacker is sandboxed in some way. An attacker cannot make

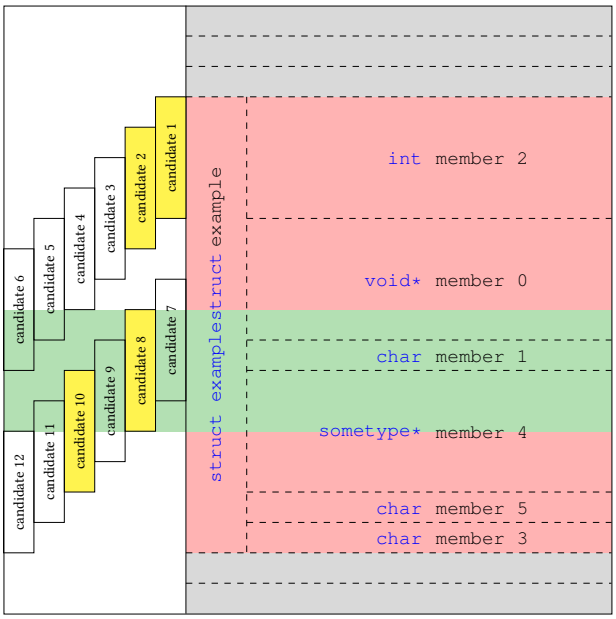

Figure 8: Visualization of offset selection for bogus argument generation from a randomized example `struct` as employed by RandCompile

targeted modifications to data structures in the Linux kernel, if the layout of the data structures is randomized at compile time.

As a side effect, this feature negatively affects forensic profile generation tools as we describe in more detail in Section 2.

There are ongoing efforts to integrate *Structure Layout Randomization* into the Clang compiler[8] to create feature parity between GCC and Clang. However, *Structure Layout Randomization* is currently only available if the kernel is compiled with GCC using the respective plugin in the kernel source tree.

B OFFSET CANDIDATE CONSTRUCTION

Our bogus argument generation method (explained in Section 3.5), uses a two-stage pseudorandom offset selection method that we visualized in Figure 8.

In the first stage, we select a set of n valid offsets within a given data structure to represent a valid memory location from which a value the size of a pointer can be sourced. In the example from Figure 8 these correspond to the subset of valid offsets marked in yellow on the left-hand side. The parameter n needs to be set at compile-time and should be chosen depending on the number of registers used to pass parameters to functions according to our calling convention as well as the average number of pre-existing parameters per function call. To keep this pseudorandom selection consistent for a given member across all instances of a given `struct`, we use information specific to this `struct`- and member type to deterministically modify the random seed. This step is important to ensure that the number of used offsets is limited to the predetermined amount across all function calls and compilation units.

In a second step, we then select one offset from the n potential choices selected in step one and use it to determine the memory location relative to the `struct` base that can be dereferenced to initialize our bogus parameter. The dereferenced value (see *candidate 8*, Figure 8) may not align with actual `struct` members, but must be contained entirely within the `struct`. Since we are selecting from the already reduced subset this time, we do not want

[8] https://reviews.llvm.org/D121556

Member used for Task Listing Analysis	Base	RandCompile (-printk, -memref)	RandCompile (-printk)	RandCompile (full)
tasks				
# most votes	3	2	2	1
# second most votes	3	1	1	-
# correct votes	3	2	2	1
# total votes	9	3	3	1
# offsets voted on	3	2	2	1
pid				
# most votes	12	6	7	4
# second most votes	-	-	-	-
# correct votes	12	6	7	4
# total votes	12	6	7	4
# offsets voted on	1	1	1	1
comm				
# most votes	33	23	28	10
# second most votes	1	1	1	1
# correct votes	33	23	28	10
# total votes	35	24	29	11
# offsets voted on	3	2	2	2
mm				
# most votes	91	9	14	10
# second most votes	4	2	10	9
# correct votes	91	9	10	10
# total votes	105	15	43	32
# offsets voted on	9	6	13	11
pgd				
# most votes	27	15	17	14
# second most votes	-	-	-	-
# correct votes	27	15	17	14
# total votes	27	15	17	14
# offsets voted on	1	1	1	1

Table 4: Detailed view of results for Task Listing Analysis: Comparison of votes received for most voted, second most voted and correct member, as well as number of different offsets voted on for each member

to consistently do so across various function calls. Thus, the information used to deterministically modify the random seed is based on function-specific information during the second step.

This creates a model that allows us to easily adapt the frequency of bogus offsets for function parameters to our specific needs by choosing n accordingly: A lower n increases the frequency of appearance of any given bogus offset but decreases the overall number of *wrong* offsets encountered by Katana. This may increase the feasibility of trial- and error approaches to obtain the correct offsets, but makes wrong offsets more likely to prevail during the majority voting procedure used by Katana to select the correct one. A high n, on the other hand, increases the number of potential offsets available to Katana but risks bogus offsets degenerating into random noise due to their low frequency of appearance when compared to the correct one.

For AMD64-based systems, $n \leq 5$ appears to be a reasonable choice, since at most 6 registers are used to pass parameters to called functions. This means that in a program with only single-parameter function calls, a total of five unused registers are available to serve as storage for bogus parameters. Accordingly, in this simplified scenario, we could set n to 5 to present Katana with 6 offsets appearing equally often, only one of which being correct. In a more realistic scenario, we would usually have an average number of function parameters that is greater than one, thus decreasing the number of unused registers available to be filled with bogus parameters. Hence, setting n to 3 or 4 is more reasonable depending on the specific configuration of the program at hand.

Looking for Honey Once Again: Detecting RDP and SMB Honeypots on the Internet

Fabian Franzen
Technical University of Munich
franzen@sec.in.tum.de

Lion Steger
Technical University of Munich
lion.steger@tum.de

Johannes Zirngibl, Patrick Sattler
Technical University of Munich
{zirngibl,sattler}@net.in.tum.de

***Abstract*—Honeypots are a widely used technique to observe the spread of malware and the emergence of new exploits. Attackers try to avoid connecting to honeypots as they reveal the attacker's methods, tools, and exploits.**

While different honeypot implementations have been fingerprinted in the past, we see a lack of studies covering Windows-related protocols such as Remote Desktop Protocol (RDP) and Server Message Block (SMB) honeypots. However, these protocols have seen at least two major security vulnerabilities in the past 5 years and are commonly exploited.

We adapted existing fingerprinting algorithms to allow an accurate identification of RDP and SMB honeypots checking how implementations behave in error conditions. We present a new improvement, namely the inclusion of system TLS stack features previously not used for honeypot detection. We are the first to perform an internet-wide scan searching for RDP and SMB honeypots. We are able to effectively uncover the presence of two common open-source honeypots for RDP and SMB each.

We identified 84 instances of *Heralding* (RDP), 1123 instances of *RDPY* (RDP), 60 instances of *Impacket* (SMB), and 1461 instances of *Dionaea* (SMB) during our scans. Furthermore, we found several hosts, which do not use Microsoft's *SChannel* TLS stack, but advertise themselves as Windows machines. This indicates the presence of a Man-in-the-Middle (MitM) box and could be a sign of a honeypot. Eventually, we analyzed how attackers interact with detectable honeypots. We deployed instances of RDP honeypots ourselves and found that credential guessing attackers seem to avoid them.

This proves that RDP and SMB honeypots are fingerprintable and that even MitM-box-based high-interaction honeypots leave detectable traces.

***Index Terms*—honeypots, Internet scanning, RDP, SMB**

1. Introduction

Honeypots have become a settled and well-researched technique to watch the emergence of exploits on different services on the Internet. They allow researchers to detect the rise of new exploits and to keep track of how widely specific vulnerabilities are being exploited.

Attackers have an interest in detecting and avoiding honeypots, as they do not want to draw attention to their specific exploitation techniques and tools [1]. Tools could be as simple as the list of passwords most often tried by an attacker, which a system administrator, in turn, may then place on a weak-password list.

Despite much research having been conducted on honeypots, security researchers still suggest new usage scenarios and designs. For example, in 2020, an improved ICS honeypot to catch malware directed at industrial control systems (ICS) [2] and a specialized honeypot for hardening neural networks against attacks [3] have been proposed.

Organizations analyzing the Internet have also shown interest. Portals like *Censys.io* and *Shodan.io* allow reasoning about the number of services on the Internet. Their data often includes information about security vulnerabilities and used software. Detecting honeypots is a logical next step. *Shodan.io* started the *Honeyscore* project to detect honeypots specifically for ICS [4] and announced they will directly annotate their search results with honeypot tags [5]. This indicates the interest of the industry in further developing honeypot detection techniques. Unfortunately, they do not share their detection methods with the public.

The Windows world has received less attention by researchers, but faced at least two wormable security vulnerabilities in the recent past. First, CVE-2017-0144 in Microsoft's Server Message Block (SMB) protocol, which is exploited by the *EternalBlue* exploit. Second, BlueKeep (CVE-2019-0708) is a vulnerability in Microsoft's Remote Desktop Protocol (RDP). Multiple security researchers reported the use of honeypots to monitor the attacks [6]. SMB and RDP protocol traffic is in the top 7 on SurfNet's Internet telescope data [7]. However, existing research primarily targets SSH, HTTP, ICS, and Telnet honeypots, but no Windows protocols such as RDP and SMB as we will discuss further in Section 2.

In this paper, we present the results of an internet-wide scan for RDP and SMB honeypots. We created a small set of network packets that allow for a fast internet-wide scan following the methods of Vetterl et al. [8] with extensions we describe further in Section 4. The set allows for an accurate classification of the hosts, as we demonstrate by performing an internet-wide scan to estimate the number of honeypots of selected open-source implementations. Moreover, we illustrate how unique features of the Microsoft SChannel and OpenSSL TLS implementation can be used to detect *abnormal* RDP stacks.

Our contributions can be summarized as follows:

(i) We present how existing honeypot fingerprinting approaches can be adapted to detect SMB and RDP honeypots. We implemented our ideas in our own *honeypot*

detector. Furthermore, we conducted a 34-day experiment to collect evidence that attackers avoid RDP honeypots.

(ii) We show that a fingerprintable TLS stack allows improved distinction between Windows and non-Windows hosts.

(iii) We performed internet-wide scans for RDP and SMB hosts and present numbers on the spread of selected open-source honeypots for these protocols.

(iv) We verified our results by connecting to a random subset of each of our classification groups and found that we have only classified 5 out of 1097 hosts incorrectly (i.e., less than 1%). In the honeypot classification groups, we have not identified a single misclassified host.

(v) We publish[1] our scanning and detection tools in order to support reproducibility such that others can use and improve upon our work. Additionally, we provide our dataset on request to interested researchers. In contrast to existing data from Censys and Shodan, our dateset includes the raw exchanged data in unparsed form and covers multiple protocol versions.

2. Related Work

Many honeypot detection approaches [7]–[10] are based on detecting shortcomings in the honeypot implementation of the respective protocol. However, the detection can also be based on the *combination of services offered* [11] or how the honeypot interacts with other services on the Internet [12]. Several open-source honeypots offer a wide set of services they can emulate. A host offering myriads of different services is unusual, especially if running services requires different operating systems. An example for interaction based detection is *HoneypotHunter* [12]. It checks for SMTP honeypots by connecting to potential open SMTP relays and sending an e-mail, addressed to a server under the detector's control. If the attacker does not receive an inbound connection on his server, especially if the host announced correct delivery, a honeypot has been found.

Vetterl et al. [8] detected SSH, Telnet, and HTTP honeypots by fuzzing honeypots and their real-world counterparts. Out of these data, they derived a *most distinctive probe* being used to perform an internet-wide scan. We based our research on this idea and will discuss it further in Sec. 4. In 2019, Morishita et al. [7] investigated the prevalence of 14 open-source honeypots in Censys Internet scanning data. They derived signatures based on protocol banners, HTTP responses, and error responses. However, they only analyze FTP, SSH, Telnet, SMTP, HTTP, and IMAP honeypots. They found about 17k honeypots for different protocols.

Honeypots and honeynets, a whole network of honeypots, have also been detected by abusing timing and network characteristics. In 2006, network packets traveling through the honeynet implementation *honeyd* were known to have a round-trip time of a multiple of ten due to characteristics of the OS and the simulation [13]. Another network latency-based detection for honeynets, which takes other fields of TCP/IP packets into consideration, is discussed in [14]. Holz et al. [1] propose the detection of so-called high-interaction honeypots, which are instrumented or sandboxed versions of the real service through artifacts of the virtualization environment or emulator.

The detection of honeypots has also found its way into the products of internet-scanning companies such as Shodan. Shodan implemented *Honeyscore* [4], which focuses on ICS honeypots and is reported to become a standard feature in their search engine [5]. Honeyscore uses a mixture of the discussed detection features. First, they consider if a service has too many open network ports. Second, they consider if a service is running in an unusual IPv4 address space (i.e., a PLC in the Amazon EC2 cloud). Third, they search if a service response matches known honeypot configurations. Lastly, they employ machine learning in an undisclosed manner for detection [2].

The interaction between honeypots and attackers has also received research interest. In 2018, Metongnon et al. [15] have analyzed data of the *SurfNet* network telescope and compared it with the incoming traffic on their telnet honeypots. In 2019, Ghiette et al. [16] conducted a large-scale study to fingerprint attackers of SSH servers by using features such the SSH banner of the client, the MD5 hash of the offered crypto algorithms and the passwords used at login attempts.

3. Background

RDP as well as SMB have a rich feature set and a long history. In the following, we introduce the basic functionality of both protocols before we outline our ideas for fingerprinting and honeypot detection in Section 4.

3.1. RDP

The Remote Desktop Protocol (RDP) allows for remote access and maintenance of the Windows operating system. Microsoft released RDP 5.1 and 5.2 together with Windows XP and Windows Server 2003 and continuously developed it ever since; now RDP 10 for Windows 10/11 and Windows Server 2016 onward is available [17]. Besides its primary screen sharing functionality, it offers additional features like clipboard sharing, redirection of peripherals such as smart card readers and drives to the RDP server. The huge amount of features inhibits a free reimplementation, even though the protocol is described as part of the Microsoft Open Specifications program [18].

Nonetheless, RDP is also used by third parties to offer remote control to non Windows operating systems. For example, a popular open-source implementation is *XRDP* for Unix-based operating systems; Oracle also offers a closed-source extension for *VirtualBox* to enable access to virtual machines using RDP.

Figure 1 shows the initial steps of the RDP protocol which are embedded in the *TPKT* format. RDP's security evolved over time and covers multiple security modes. The client and server agree upon which mode should be used during the first steps of the protocol (Step 1 in Figure 1). In the context of our work, the following modes are relevant:
PROTOCOL_RDP: This mode must be supported by the client and indicates the use of standard RDP security; a mode where encryption and integrity protection *can* be done in the RDP protocol itself. Step 2 and 3 in

1. https://github.com/tum-itsec/looking-for-honey-once-again

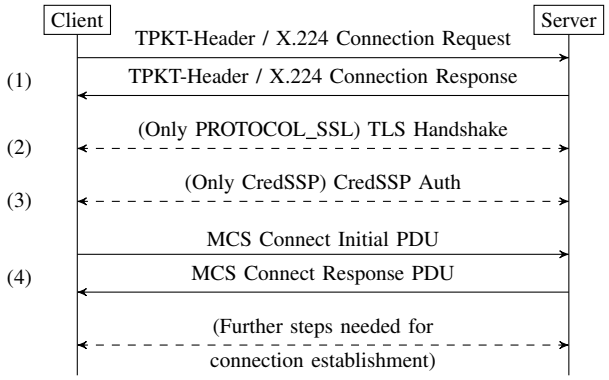

Figure 1. The first steps of the RDP protocol.

Figure 1 are skipped in this case. Most of the selectable cryptographic primitives in this mode (such as RC4 and MD5) are outdated nowadays.

PROTOCOL_SSL: After the first message exchange, the negotiation phase, both parties establish a secure channel via the TLS protocol. Authentication of the server is realized through an X.509 certificate. Authentication of the client is done by the user entering their credentials into the login UI after all other negotiations are finished.

PROTOCOL_HYBRID: The authentication of the user is done immediately after the TLS handshake has finished. All further negotiations are done *afterwards*. Microsoft refers to this mode also as *Network Level Authentication (NLA)*. Authentication of the user could be done via a classic (username, password) tuple, by passing a Kerberos ticket, or by using a smart card.

Note that only in the *PROTOCOL_HYBRID* mode the user has to authenticate itself early in the third step. In all other modes listed above, the server will willingly provide a lot of information and commit resources on establishing the connection before the user is authenticated. Therefore, recent Windows installations suggest the use of Network Level Authentication, which is equivalent to enforcing *PROTOCOL_HYBRID* during connection establishment (from Windows Server 2008 R2 onward).

3.2. SMB

The Server Message Block protocol (SMB) is primarily used by Windows operating systems to share locally stored files with other machines in the same network to allow browsing, editing, and deletion. Besides sharing files, it also offers inter-process communication, e.g., in the form of named pipes. On top of that, Microsoft Remote Procedure Calls (MSRPC) can be used.

Microsoft released SMB 1.0 together with Windows 2000 and SMB 3.1.1 with Windows 10. Windows supports end-to-end encryption and integrity protection since SMB 3.0. Beforehand, SMB offered none of these services, rendering it an easy target to attackers and making the protocol unsuitable for communication over an untrusted network such as the Internet. Therefore, many network operators filter the default SMB port 445. Nevertheless, around 1M hosts are offering their SMB service on the Internet according to Censys data.

Besides the implementation in Microsoft Windows, *Samba* has become the most prominent open-source implementation of SMB. We chose two versions, 3.5.6, as it is the last version of Samba without support for SMBv2 and 4.10.0, being the latest version available to Ubuntu 19.10 at the time of our experiments. Furthermore, commercial competitors such as Visuality Systems have also implemented SMB.

SMB communication starts by negotiating a dialect (e.g., “NT LANMAN 1.0” or “SMB 2.002”) with the server. If backward compatibility is required, the negotiation is started by sending an SMBv1 negotiation packet. A list of supported dialects is sent alongside this negotiation packet, similar to TLS. If backward compatibility is not desired, a connection can also start with an SMBv2 packet. The server selects one dialect from the offered list and responds with its choice.

After dialect negotiation, the sequence continues with the Session setup phase. During this phase, the user is authenticated via the Simple and Protected GSSAPI Negotiation (SPNEGO) that uses the General Security Service API (GSSAPI) and which in turn transports authentication mechanisms such as Kerberos or the NT Lan Manager Protocol (NTLM). The server can choose to authenticate the user without credentials and grant access to the server anonymously.

3.3. Honeypot Implementations

Honeypot implementations are grouped into *low*, *medium* and *high-interaction* honeypots. Low- and medium-interaction honeypots focus on easy deployment and maintenance while implementing only basic functionality. High-interaction honeypots try to mimic a service indistinguishable from the original and are often based on virtual machines to minimize operational risks.

We focused on finding open-source implementations for RDP and SMB mentioned above. We identified two honeypot implementations respectively, which we will describe in the following.

***Heralding* (RDP)** *Heralding* [19] focuses on catching login credentials of everybody logging into the system. Therefore, it is not in focus of the developers to provide a complete protocol implementation. RDP connections are terminated early if the credentials are not submitted during connection establishment.

***RDPY* (RDP)** Another honeypot implementation for the RDP protocol is *RDPY* [20], written in Python 2 on top of the Twisted framework. The first commit dates back to the year 2013. Since then, it has been heavily developed until 2015, but no RDP protocol related fix happened afterwards. *RDPY* supports `PROTOCOL_SSL` and `PROTOCOL_RDP` security. However, if the administrator does not create an X.509 certificate, `PROTOCOL_SSL` will be disabled. In contrast to *Heralding*, an attacker can complete the full RDP connection sequence and will be able to watch an emulated screen. During operation as honeypot, content and events on the emulated screen are replayed from a session recorded beforehand. In order to create a recording, RPDY is used in a preparation phase as a proxy to a regular server. The implementation proxys the requests to the regular server and records all screen update and keypress events. Afterwards, the recording is replayed to every attacker. This allows the honeypot to

TABLE 1. FEATURES OF ANALYZED HONEYPOT IMPLEMENTATIONS

Honeypot	Age	RDP			SMB		
		RDPSec	SSL	Hybrid	v1	v2	v3
Impacket	2003				✔	✔	✔
Dionaea	2009				✔		
RDPY	2013	✔	✔				
Heralding	2012		✔				

mimic a real system as long as the attacker does not try to interact with the system.

***Dionaea* (SMB)** *Dionaea* [21] is a honeypot for a wide range of protocols. Since its initial git commit back in 2009, it advertises support for over 14 protocols, including FTP, HTTP, Memcached, MSSQL, SMB, SIP and UPNP. The *Dionaea* core is implemented in the C programming language with support for Python modules implementing most of its protocol logic. *Dionaea* does not offer RDP support. SMB support is implemented in Python and restricted to SMBv1.

***Impacket* (SMB)** *Impacket* is a collection of Python classes for working with networking protocols especially from the Windows domain. This includes high level implementations of SMBv1, SMBv2, and SMBv3 [22]. While *Impacket* is not a honeypot on its own, we found honeypot implementations using this library during our Internet scans. The library is provided by SecureAuth, a security company, and seems to be solely developed for the needs of penetration testers. SecureAuth provides numerous examples how to use the library in this field, e.g., an exploit for the SMB Relay Attack (CVE-2015-0005).

Besides the honeypots mentioned, further implementations targeting SMB are for example *DTK* or *Smoke Detector* (see also [23] for a more complete survey of popular honeypot implementations). However, these implementations have stalled in their development since 2005 and we will therefore not consider them here.

Table 1 summarizes the different characteristics of selected honeypots. These honeypots can also be combined to cover multiple protocols or scenarios. An example are the T-Pot Docker and virtual machine images created by T-Systems, which also hosts T-Pot actively for their Sicherheitstacho project[2]. It contains all presented tools but *Impacket*.

3.4. TLS Fingerprinting

As our approach will leverage the fact that many honeypots implement their services without taking the TLS stack into consideration, we provide a short overview of TLS fingerprinting techniques. We found that modern RDP implementations mainly use TLS 1.2. Therefore, we will not describe the specifics of the more recent TLS 1.3.

During the establishment of a TLS connection, the cipher suite for encryption and integrity protection as well as the algorithms for key exchange need to be negotiated. In addition, the client can present a set of supported TLS extensions. The server decides which extension subset is accepted and signals this together with the selected cipher in the *Server Hello Message*.

2. https://www.sicherheitstacho.eu/start/main

This property was used by research [24], [25] and different groups to fingerprint TLS clients. Notable tools are JA3 [26], Cisco Mercury [27] and Cisco Joy [28]. They are based on similar approaches, creating a hash of the mentioned parameters of the TLS Client Hello to identify implementations. JA3 provides an additional extension JA3s to fingerprint servers besides clients. The server behavior is not only influenced by the used implementation but also by each Client Hello because it can only select offered properties for a successful handshake. Therefore, JA3s relies on the combination of client and server fingerprints. In order to proactively get server fingerprints, JARM [29] was developed. It fingerprints a server based on its behavior for 10 manually crafted Client Hellos.

The approach presented in this paper is based on our observation of two further characteristics. First, if the server agrees on a PFS-enabled cipher, an algorithm for key exchange needs to be negotiated. We observed frequent use of ECC here. If chosen, the curve needs to be agreed upon before the key exchange can happen.

Second, TLS allows multiple handshake messages to be packed together inside a single TLS record packet. The *SChannel Service Provider* (SSP) implementation of Windows packages does this. In contrast, OpenSSL creates a TLS record for each handshake message.

4. Our Honeypot Detector

SMB and RDP are both complex protocols that are very hard to implement identical and feature-complete with respect to a reference implementation when developing a low- to medium-interaction honeypot.

Following the methods of Vetterl et al. [8], we fuzzed all RDP and SMB implementations we are aware of to find the most distinctive probe. In an optimal case, a single packet can be used to classify all hosts. In contrast to Vetterl et al., we use *a set of distinctive packets* to increase the resilience of our scanner to unknown implementations. Our request set is still small enough to enable scan speeds high enough to perform an internet-wide scan.

Where applicable, we also fingerprint the TLS stack used by the respective implementation. Furthermore, our fingerprinting tool does not consider the specifics of the operating system TCP stack. Some fingerprinted implementations also support other operating systems such as Linux.

We created two separate tools to create and compare fingerprints of *known* RDP and SMB implementations respectively. We created the fingerprints on our lab instances of all mentioned honeypots and benign Windows installations. Afterwards, we tested the fingerprints on real-world instances of Windows Server 2012, 2016, 2019 in the Amazon EC2 cloud and Microsoft Azure. We found that the Windows end-user versions share RDP and SMB server code with the server versions (see Table 7).

4.1. Packet Similarity Scoring

In order to check for the similarity of the collected packet exchanges, we developed a parser for the SMB and RDP protocol which tokenizes a single response

into multiple labeled fields. The fields of two responses are compared by checking if their values are identical, in a certain range or have certain bits enabled. Fields containing random values, length indicators (where different lengths are allowed), timestamps or fields containing configurable values are ignored during comparison. The similarity score is eventually set to be $\frac{1}{1+n}$ where n is the number of differing fields. Thereby, the similarity decreases the more fields inside the responses are found to differ.

One notable exception is the parsing of TLS traffic occurring inside RDP traffic. For the sake of simplicity, we only partially parse the TLS traffic and convert it into a single synthetic tokenized packet. This synthetic packet contains all relevant characteristics of the TLS handshake mentioned in Section 3.4.

4.2. Differential Fuzzing for Fingerprinting

Equipped with a metric to calculate the similarity of two packets, we can build a small fuzzer upon these. The fuzzer constructs a packet, sends it to each implementation, and utilizes the similarity scoring to decide if a notable difference between the answers exists. The responses are then compared to each other according to the similarity metric mentioned above. The request packet generating the set of responses least similar to each other is selected as the most distinctive probe.

For RDP, we utilize a bit mutation strategy of the first packet. For SMB, we construct the set of all reasonable combinations of values inside the request packet fields to find the most distinctive probe. While this fuzzing approach is simple, it is sufficient to identify fields in the protocol causing distinguishable answers for each implementation.

4.3. Our RDP scanner

Based on out fuzzing, the protocol field with the most notable impact on the server responses is the proposed security mode of the client (see Sec. 3.1). Therefore, we establish four connections with the target host. Three of them are regular connection attempts, advertising all major security modes (`PROTOCOL_RDP`, `PROCOTOL_SSL`, and `PROTOCOL_HYBRID`). This is done because some implementations choose to downgrade from or directly refuse connection attempts advertising support for certain security modes. The fourth connection contains an invalid set length field, triggering different error handling behavior between *Heralding* and *RDPY* and all Windows versions. Regular Windows versions close the connection with a Connection Reset. *RDPY* closes the connection by sending a regular TCP FIN packet to the client and *Heralding* sends the client an RDP Negotation Failure Message.

Still, in many cases just collecting the first packet exchange for each connection does not provide enough information to differentiate between implementations. Therefore, depending on the negotiated connection type we try to either establish a TLS connection and send the next packet over TLS or send it unencrypted. Note that the TLS fingerprint of *Heralding* and *RDPY* is not affected if the implementations are operated on a non-Linux operating system. Both packages rely on OpenSSL and do not switch to the native operating system TLS stack.

If the server accepts our `PROTOCOL_HYBRID` attempt, the protocol proceeds with Network Level Authentication (NLA) after the TLS handshake. We send a regular SPNEGO packet, starting the exchange of security mechanisms of both parties, but abort the connection directly afterwards in order to avoid supplying credentials for ethical reasons (see Sec. 5.1).

In case of a `PROTOCOL_RDP` or `PROTOCOL_SSL` connection, we proceed with the second step and send an encrypted or unencrypted version (dependent on the security mode setting) of the client capabilities which we recorded as well from the standard Windows RDP client.

We found that all Windows Servers which can be rented in the Amazon EC2 and Azure cloud have NLA enabled by default.

Because of the limited information we are able to obtain in an NLA enforced environment, it is impossible to distinguish Windows 10, Windows 8 and the different versions of the Windows Server 2012, 2016, 2019 solely on the characteristics of exchanged RDP packets.

The flag field of the RDP Negotiation Response PDU has two flags which are only available in recent RDP versions. If NLA login is not enforced, we can find more distinguishing features in the second round trip of the protocol. We show an example of which fields can vary in the responses of the second round trip in Table 6. It shows the relevant parts of the reference responses to the `PROTOCOL_RDP` packet. Packet headers that do not contribute to the fingerprint are intentionally left blank. Some implementations, e.g., Windows 10, do not respond to the second packet because they discontinue the connection after receiving it, which also counts as distinguishing behavior.

4.4. Our SMB scanner

We found the following fingerprintable implementation pitfalls for SMB: If no common dialect can be negotiated between client and server, different implementations have different ways of discontinuing the connection, some sending error codes while some close the connection without sending additional data. Furthermore, our fuzzer found different behavior by setting reserved fields to non-zero values or by setting other fields to values that are not reasonable in the current context. For fields inside the request which have corresponding fields in the response, it is up to the implementation whether such atypical values are ignored by mirroring them or by resetting them to zero.

Lastly, another valuable factor contributing to the fingerprint of an implementation is the combination of the *Native OS* and *Native LAN Manager* fields inside the second packet in an SMBv1 protocol sequence. It can offer hints about the platform and operating system or environment which the server runs on and while honeypots can trivially replace this information, it does contribute to the overall fingerprint.

Therefore, our final SMB scanner establishes four connections to a scanned host, each having a different function:

1) An SMBv1 packet checking if the targeted host supports SMBv1 (as we consider this to be a requirement for a honeypot, see Sec. 5.5)
2) A packet that checks if the targeted host supports anonymous login
3) A packet checking for the support of the most recent version SMBv3
4) A specifically crafted packet that we found to trigger the most distinctive responses from implementations that we consider in our work

5. Internet-wide Scan

In order to test our ideas and developed tools, we conducted an internet-wide scan. We will describe its execution and its results in this section.

5.1. Ethical Considerations

We follow the principles of informed consent [30] and best practices [31]: we avoid the collection of personal or sensitive data and we try to avoid causing any harm to online servers during our active scans. None of the scan probes we send have affected our test machines negatively.

As described in Section 4, both the RDP and SMB protocol usually provide enough data for fingerprinting before we have to actually login into the machine. Since we do not provide any credentials to our targets, we do not consider it as a hacking attempt.

We took precautions to minimize the impact of our scans, following established practices as, for instance, described in [32]. In particular, we maintain a block list to avoid scanning systems that have in the past indicated to us that they want to be excluded from scans. Our abuse email address is published in the WHOIS and all abuse emails are forwarded to us by our IT department. We assess the impact of our scans in terms of potential harm to other systems and human beings, as proposed by the Menlo report [30]. We use a relatively low scanning rate to minimize any impact and respond immediately to complaints. All our scanning hosts run a web server which provides information on the scan including an email contact. We received 9 new requests regarding our scan activities, added IP addresses of eight requests to our block list and resolved one request, allowing us to continue scans as research project.

5.2. Setup

For our Internet-wide scan to find honeypots, we combine the implemented detectors with ZMap [32]. ZMap provides us the possibility to scan the complete IPv4 address space searching for hosts with open RDP (3389) or SMB (445) ports. If a host with an open port is detected, it is directly handed to the respective honeypot detector. This combination of ZMap as a stateless, fast Internet-wide scanning tool and our implemented stateful detector allows fast but informative scans.

Due to the properties of ZMap to focus on a single protocol and port to drastically decrease scan duration, we scanned SMB and RDP sequentially. Each scan was conducted with a rate of 20 000 packets per second to reduce the impact on target systems. For each scan, we use an up-to-date BGP dump from our local upstream provider with the complete set of reachable prefixes as a ZMap allow list to reduce the scan duration. This excludes all address ranges from the scan that are not announced by any AS and thus not reachable. Furthermore, we use a self-administered block list to prevent scanning targets which requested to be excluded from our scans. This block list was created over time based on abuse mails received by our research group to follow the ethical considerations described in Section 5.1. The list is solely built from requests to be excluded from active scans not including any external sources. The block list contains different addresses and prefixes covering 5.7M addresses in total. With the given setup and rate, each scan probes 2.8 billion IP addresses (67% of the complete IPv4 address space) and has a duration of around 37 hours.

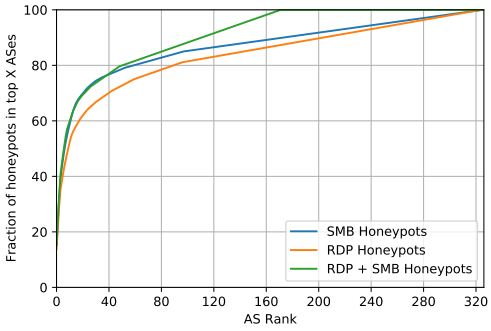

Figure 2. AS distribution of honeypot addresses

5.3. Classification Results

During our internet-wide scan started on March 26th 2021 we discovered in total 7.5M hosts with an open RDP port. We could successfully assign 2.1M hosts to known RDP implementations. Regarding SMB, we found 2.7M hosts with an open port 445 during the scan started on the March 30th 2021. 1.1M hosts were categorized after a successful connection could be completed. For both protocols, we only label a connection with a specific implementation if the fingerprint matches *exactly*. Results can be seen in table 2 and we will describe the results for both protocols separately.

56.7% of the RDP scans and 57.2% of the SMB scans resulted in an error. For both protocols more than 97% of the errors were caused by a closed connection. Either a host is not reachable at all or it closes the connection after seeing our initial RDP connection attempt. This means most likely that those hosts do not provide the respective service behind the scanned port or that the port was erroneously reported by ZMap. In the remaining cases (less than 3%), we are not able to parse the answer of the host. We assume this is due to missing functionality in our parser or the host offers a different service. 14.2% of RDP hosts and 1.1% of the SMB hosts are not categorizable. Hosts are considered as not categorizable, if they do not match one of our recorded fingerprints.

RDP We were able to find 1207 matches with the *Heralding* and *RDPY* honeypots with 100% similarity. Predominantly, we found instances of *RDPY* in a configuration

TABLE 2. SCAN RESULTS OF OUR INTERNET WIDE SCAN.

Category	RDP	SMB
Total ZMap results[1]	7 577 919	2 704 250
Categorizable[2]	2 125 428	1 125 838
Regular Implementations	1 940 159	1 124 317
Windows 8 & 10[3]	1 401 465	96 361
SChannel	1 401 452	
non-SChannel	13	
Windows 10 (no NLA)	96 003	
SChannel	96 003	
Windows 8 (no NLA)	35 126	
SChannel	35 126	
Windows 7	357 179	685 701
No Data	296 065	
SChannel	61 113	
non-SChannel	1	
Windows XP	36 823	3834
No data	36 823	
XRDP	13 355	
non-SChannel	13 355	
VirtualBox	208	
No data	208	
Samba 3.5.6		153 071
Samba 4.10.0		181 931
YNQ		2741
Misc. implementations[4]		678
Honeypots	1207	1521
Heralding	84	
non-SChannel	84	
RDPY	50	
OpenSSL	50	
RDPY (no TLS)	1073	
No data	1073	
Dionaea		1461
impacket		60
Non RDP/SMB protocols	245 300	
HTTP	185 948	
SSH	59 352	
Uncategorizable[2]	1 080 773	31 152
Errors	4 310 480	1 547 260
Unparseable	11 649	36 395
Connection closed[9]	3 127 932	1 419 216
No connectivity[8]	1 170 899	91 649

[1] ZMap only reports hosts with an open port. An open port is no proof the respective service is also provided.
[2] Hosts are only labeled with a classification if the fingerprint shows an *exact* match.
[3] and Windows Server 2012R2, 2016 and 2019
[4] Combined set of different rare implementations.
[5] Unknown Non-RDP protocol
[7] Hosts which did not respond to all of our packets preventing an exact classification
[8] Hosts without any response to our scanner despite being reported by ZMap
[9] Hosts which we established a connection with but closed the connection before sending any data

TABLE 3. TOP 10 AUTONOMOUS SYSTEMS HOSTING HONEYPOTS

CO	ASN	Organization	SMB	RDP	**Total**
US	16509	AMAZON	232	167	399
US	20473	CHOOPA	126	95	221
US	14061	DIGITALOCEAN	102	90	192
DE	197540	netcup	66	72	138
TW	1659	TANet	131	1	132
US	8075	MICROSOFT	48	25	73
US	63949	Linode	33	37	70
US	14618	AMAZON	41	28	69
US	15169	GOOGLE	35	32	67
US	22773	Cox Communications	50	3	53

not offering `PROTOCOL_SSL` and always falling back to standard RDP security. Furthermore, we observed that the RDP port is frequently used for non RDP services. If a response packet of a host does not appear to be a valid TPKT, we classified it as non-RDP. Investigating this response class, we found several hosts answering with `HTTP/1.1 400 Bad request` indicating an HTTP server. Moreover, we found many SSH banners sent in response to our probe indicating an SSH server.

We found the RDP stacks of Windows 8 and Windows 10, which are also used in Windows Server 2012 and 2019 respectively, predominantly used. In about 1.4M cases the hosts enforce the use of Network Layer Authentication (see Section 3.1) which gives our scanner only limited opportunity for fingerprinting. In almost all cases, the Windows hosts have a fingerprint which indicates the use of *S-Channel* the Microsoft Windows TLS implementation. However, a few hosts show perfect match with Windows, but have a fingerprint indicating the use of a non-Microsoft TLS library.

We also found Windows 7 and XP hosts unwilling to negotiate a TLS based connection with our scanner, but downgrading us to Standard RDP security for unclear reasons. This is marked in Table 2 as *No Data*.

SMB 1521 hosts have responded to our probe packets in exactly the same way as our lab instances of *Dionaea* and *Impacket*. With 60%, a majority of hosts that can be classified use Windows 7 followed by Samba 4.10.0 (16.3%) and Samba 3.5.6 (14.3%). Further analyzing the set of values we found inside the *NT Lan Manager* field, we discovered numerous rarely appearing and some almost undocumented implementations of the SMB protocol such as smbx (Apple), Alfresco CIFS and SXLM for which we had no fingerprint from lab tests. Since we did not expect any honeypot to mimic them, we categorized them as *Misc. implementations* without further comparison of their fingerprint.

RDP + SMB The two IP address sets of identified RDP and SMB honeypots are not distinct. With 606 mutual addresses, around 40% combine honeypots for the RDP as well as for the SMB protocol on the same host. This high overlap supports our finding that the presented methodology is able to detect honeypots. Selective searching of the hosts on portals like Censys.io or Shodan.io reveals that these hosts usually offer services associated with the honeypots *Dionaea* and *cowrie*, a honeypot for SSH and Telnet. We found eight hosts that mix an RDP honeypot with an *Impacket* installation, supporting our argument that *Impacket* is indeed used for building up honeypots.

Furthermore, hosts of 36 RDP honeypots have an open SMB port which can not be classified as a honeypot. Out of these, 35 SMB connection attempts ended in a premature session exit, indicating an unresponsive service. The one remaining host is classified as *uncategorized*, however the handshake sequence has some similarity with Windows XP, which could indicate an unknown honeypot implementation. Vice versa, 87 SMB honeypots additionally have an open RDP port reported by ZMap. In 77 cases our scanner was not able to successfully establish a RDP connection, in eight cases a non-RDP service answered our probe and two hosts have not been classified.

5.4. AS Coverage

To highlight the widespread usage of detectable honeypots by a multitude of different organizations and providers, we analyze the distribution of honeypots across autonomous systems (AS). Figure 2 shows the distribution of all identified honeypots across ASes. 50% of all detected SMB honeypots are located on only 12 ASes but the remaining 50% are spread across 314 ASes. RDP honeypots are located in more ASes with a total coverage of 325 but with 50% located in 10 ASes, a majority can again be found in a small subset. Table 3 lists the top 10 ASes based on the total amount (SMB and RDP) of honeypots. It can be seen that multiple listed organizations are mainly cloud providers, (e.g., Amazon, DigitalOcean, and Netcup) and research networks (AS1659 is the Taiwan Academic Network). While some honeypots might be hosted by the cloud hosting organizations themselves, a majority is most likely set up by the provider's customers. Some of these hosting providers are also known to be scanning and research friendly. That might be an explanation for smaller hosters appearing in this top list. Based on these results, we infer that these types of honeypots are a widely used tool across various ASes.

5.5. Result Validation

In the following, we will describe our attempts to validate our scanning results by using additional indicators, because of the lack of ground-truth data for the Internet. We already observed in Section 5.3 that hosts offering both analyzed services are highly likely to have both services classified as honeypot. This is a clear indicator that these hosts are honeypots.

Additionally, we automatically connect to hosts of each category using established open-source tools and collect their diagnostic data where possible. If an automatic validation is not possible for a category, we fall back to manual methods. Table 4 summarizes our validation results. Unresponsive machines were most likely taken offline during the time period between scan and verification. We miss-classified less than 1% of responsive hosts. Additionally, *none* of the hosts in the honeypot group was miss-classified.

5.5.1. RDP Validation. First, we check if hosts being classified as normal Windows machines have been labeled with the correct Windows version. To achieve that, we utilize *rdesktop*[3] to obtain a screenshot of 100 target hosts in each category in an automated way. This is only possible for hosts not enforcing NLA. The captured screenshots of the login screens can then be quickly checked by using an image recognition algorithm or a human analyst if they match the respective version of Windows. The results are shown in Table 4, only one Windows 10 host is misclassified.

Second, we check if the honeypots have been labeled correctly by our scanner tooling. Unfortunately, the screen scraping approach is not viable to detect the selected honeypots. As *RDPY* only replays pre-recorded sessions, a human analyst will quickly notice that keypresses are not displayed and that he is viewing how somebody else is using the machine. However, to a screen scraping tool, the host will appear as a benign machine. *Heralding* is unable to go through the whole connection sequence and will terminate the connection early and never continue to a point where a user can see the Windows login screen. Therefore, we connect to 30 instances of each honeypot category manually and analyze the exchanged packets and check if the machine reacts to mouse movements.

During our manual verification process, we observed that many of the *Heralding* honeypots use the same subject and issuer names in the TLS certificates and that they show the abnormal protocol behavior described above. This strengthens our assumption that these hosts are indeed *Heralding* honeypots. Multiple *RDPY* instances share desktop session replays they present to the connecting user. Foremost, we observed a login screen of Windows 8/10 fading in, but a few instances presented us with a full Windows 7 desktop session where somebody moves the mouse on the desktop while our test user did not interact with the system.

We also performed manual validation on the small set of non-SChannel hosts being classified as regular Windows instances. As our classifier showed good precision for regular hosts, we believe that a real windows host is involved in the connection. We conclude that the connection is interrupted by a MitM-box. The non-SChannel Windows machines appear to be fully functional. Dialogs, for example, the accessibility menus, react normally.

5.5.2. SMB Validation. We connect to each host using the tool *smbclient*[4], which is capable of showing the offered file shares of an SMB server. We observed that the shares offered to a client by *Dionaea* are configurable in the most recent versions, but we assume many users might not change the default values. We connected to all *Dionaea* honeypots and found that 628 hosts have not switched the offered default shares and displayed comments. Nine hosts presented a different list of file shares. We manually connected to these instances and found that the share names are only slightly altered and that the connection dies with unusual error codes when we

TABLE 4. VERIFICATION RESULTS

	Correct	Incorrect	Unresponsive / Error
RDP	**448**	**5**	**127**
Heralding [1]	29	0	1
RDPY [1]	29	0	1
RDPY (no TLS)[1]	18	0	12
Windows 10 (no NLA)	89	1	10
Windows 8 (no NLA)	77	0	23
Windows 7 (no NLA)	58	0	32
Windows Server 2003	55	4	41
XRDP	93	0	7
SMB	**649**	**0**	**863**
Dionaea	628	0 (9)[2]	824
Impacket [1]	21	0	39
Total	**1097**	**5**	**951**

[1] Only manually verified
[2] Manual additional test deemed all hosts as honeypots

3. https://github.com/rdesktop/rdesktop

4. https://www.samba.org/samba/docs/current/man-html/smbclient.1.html

browse around. For the validation of *Impacket* instances, we observed that the amount of free disk space that is displayed to the user while browsing through offered files is a hardcoded value in the source code. Therefore, we employ this as a detection feature.

6. Attacker Behavior Analysis

To study how attackers react to detectable honeypots and how they adopt their behavior, we deployed multiple instances of RDP honeypots at various cloud providers. We captured their traffic and analyzed the performed attacks.

6.1. Setup

Our honeypots were deployed in the Amazon EC2 cloud from June 17 to July 19, 2021 (34 days). To reduce the influence of the previous owner of the IP address we received from Amazon, we assigned a new IPv4 address obtained from the Amazon EC2 pool to each of our honeypots every 7 days. All traffic was captured and recorded in PCAP files using the EC2 infrastructure during our experiment. Because RDP is already the default remote acccess protocol for Windows machines in the EC2 cloud and our scans identified more open RDP servers than SMB servers we decided to only set up RDP honeypots. For the comparison, we use an unmodified Windows Server 2019, an instance of *RDPY* (the most common RDP honeypot detected during our scans) and two instances of *PyRDP* [33], a pure MitM honeypot for the Windows RDP protocol (one in the default configuration, and one in "no downgrade" (ND) mode to decrease the detection surface).

To analyze the potential impact of the high concentration of honeypots in the autonomous systems of a few cloud providers, we rented a single Linux host in AS 197540 (netcup), AS 14061 (DIGITALOCEAN) and AS 6724 (STRATO) each, and set up a host in a research network, namely AS 209335 (TUM). On the rented hosts, we use `iptables NAT` to transparently forward the RDP traffic to our EC2 installation. In order to evenly spread the traffic from these proxies over the EC2 honeypots, each incoming connection is sorted into a hash bucket (`iptables HMARK`) based on the source IP of the client. Therefore, the IP appears to be a different honeypot depending on the client IP. However, as long as a client keeps its IP address, it will always connect to the same honeypot instance.

6.2. Results

We found that our honeypots seem primarily subject to credential guessing attacks, where an attacker tries out different well known usernames and weak passwords.

Fig. 3 visualizes the distribution of incoming connections between our machines. We categorized the incoming traffic for each honeypot into three categories: (1) *RDP traffic*, which is indicated by a TPKT header starting with a `0x3` byte, (2) an *empty request*, which is a completed TCP three-way handshake resulting in a connection that is immediately closed afterwards, and (3) *non RDP traffic*,

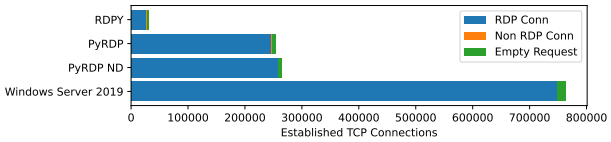

Figure 3. Traffic distribution between our honeypot implementations

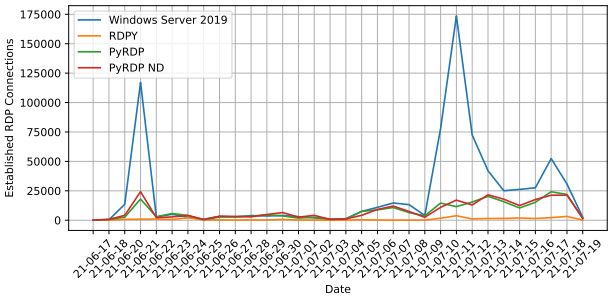

Figure 4. Traffic distribution between our honeypot implementations

where a client sends a non-empty request not matching the criteria of (1). However, all honeypots received less than 1% of (2) *empty requests* and less than 0.2% of (3) *non-RDP traffic* on the RDP port. We removed these connections from the dataset prior to analysis. We observed connections from *Censys.io* and *Shodan.io* of type (2) and we expect the remaining to be opened by lesser known Internet scanning services.

Our non-honeypot Windows Server 2019 seems to be preferred by attackers over our remaining honeypots. Fig. 4 illustrates the number of RDP connections to our honeypots over time. The traffic originates primarily from three attacks on our honeypot infrastructure on June 20, July 10, and July 16, 2021.

The first attack on 20th of June is mainly caused by a massive amount of connections originating from a /24 IP subnet from a single autonomous system (ASN 49505, Selctel, located in Russia). The honeypots on the Amazon EC2 instance were attacked directly via their respective IPs and not via one of our forwarding proxies.

About 40% of the connections established on July 10 originate from two IPs that are likewise located in the same AS and that have not communicated with any other honeypot. Furthermore, about 6% of the connections were proxied from AS 197540 (netcup) on this day and spread almost evenly across all honeypots with the Windows Server 2019 in a slight lead.

The traffic peak on July 16 consists of about 23% out of connection attempts from ASN 209335. Despite proxied traffic being balanced across targets, most connections (63 %) were made to the Windows Server 2019, followed by *PyRDP* (13%) and *RDPY* (1%). Unfortunately, our proxy setup does not allow us to further trace the connections back to the original IPs of the attacker.

The data suggests that honeypots are indeed avoided by attackers even if they are just doing credential stuffing attacks. We observed that multiple hosts in an IP address range seem to collaborate, i.e., a host *A* sends an initial probe, terminates the connection and processes our answer internally. Afterward, host *B* gets notified by *A* if the host is of interest and performs further scanning or an

attack. This pattern is provably present for probes we received from *Censys.io* as the scanning machines have meaningful RDNS records and the the targeted attacks on two honeypots on July 10 suggests that attacking host act likewise. For a more in depth analysis, this experiment needs to be repeated at larger scale.

7. Discussion

Our analysis relies on active internet-wide scans and is therefore affected by inaccuracies due to the heterogeneity of the Internet and the impact of a single vantage point. Data from internet-wide scans always shows a certain degree of noise due to the fact that the Internet is built from individually managed components. This impacts the data quality and classification rate as shown in Table 2. While some of the hosts reported by ZMap can not be classified due to connection timeouts or unspecified behavior, others even provide different services on the scanned port. Furthermore, internet-wide scans from a single vantage point might not be able to reach the complete Internet as shown recently by Wan et al. [34], which can affect our results.

Nevertheless, we avoid inaccuracies with strict constraints in our classification methodology. Our fingerprints ignore all configurable options in the respective implementations we are aware of. A mismatch between fingerprints therefore indicates a difference in implementations, rather than in configurations. On the one hand, this allows us to provide a meaningful and accurate lower bound of existing honeypots spread across a multitude of ASes and organizations. On the other hand, we consider many hosts to be *uncategorizable*.

A few of the classifications might be caused by the heterogeneity of the Internet. Other factors include differences caused by different patch levels of the SMB and RDP implementation. In samples we took from the *Uncategorizable* group, we find a large amount of Windows hosts with varying operating system versions for which we rejected the correct classification label because of our strong matching criteria. However, we believe this does not impair our ability to find known honeypot implementations as their fingerprint is quite unique and not subject to much change. The *RDPY* source code has not changed in its protocol handling for 6 years, and *Dionaea* saw its latest change in the module responsible for the handling of SMB back in 2017.

7.1. What about high interaction honeypots?

We believe that hosts that we classified as *Non-SChannel Windows RDP Hosts* are likely high interaction honeypots, where the TLS connection is interrupted by a non-SChannel MitM-box in order to observe the traffic. Unfortunately, 13 exact matches with Windows enforce NLA, so that we could not further investigate if the classification is correct without ethical issues caused by providing login credentials.

During analysis of our Internet scans, we became aware of the Wallix Redemption MitM-box for RDP[5] and the *PyRDP* [33] MitM honeypot. Both use OpenSSL for re-encryption of the traffic, with a clearly distinguishable fingerprint. The one non-NLA-enforced host could be such a *PyRDP* honeypot: It reacts to user input in a meaningful way (e.g dialog boxes and keyboard actions are processed), and allows "normal" interaction with the system (e.g. the *Disconnect* button actually disconnects the session) despite the abnormal SSL stack.

If we relax the 100% similarity constraint, we were able to find other hosts that behave like Windows machines, do not require login on connect, and are attached to the AS of DigitalOcean. To our knowledge DigitalOcean does not offer or officially support Windows machines. Furthermore, we found similar implementations in their network offering different Windows versions with the same TLS certificate. In contrast to the 13 hosts mentioned above, these hosts downgrade the protocol security level from `PROTOCOL_HYBRID` to `PROTOCOL_SSL`, which we have never observed in a unmodified Windows version during our tests, being another indicator for an abnormal RDP stack.

We believe that our data set has more interesting high-interaction honeypots in the *Uncategorized Hosts* section. However, our existing results already show that this detection method is viable.

7.2. Why should we care?

Our experiments confirm that also commonly used RDP and SMB honeypots are fingerprintable and that a fingerprint can easily be created. In Section 6 we conducted a study to check if fingerprinting techniques are applied by attackers. While the observation time of one month is not long enough for a final answer, our experiment data suggets that this is the case even if we have not observed their exact fingerprinting technique. It is also worth noting that both *PyRDP* instances (with we would classify as a high interaction honeypot) received less attack traffic, indicating that more care needs to be taken with regards to details like the *abnormal* TLS stack.

We suggest the following improvements to honeypot implementers and operators:

Employ differential fuzzing yourself. Implementers can use the presented fingerprinting methods themselves in order to minimize the observable behavioral gap between their honeypot and the original protocol implementation.

Be careful about the TLS stack implementation. If you are operating a MitM box in order to monitor encrypted TLS traffic in a protocol of interest, select a TLS implementation that matches the target machine. For example, SChannel has a standardized API on Windows, which allows third-party code to use it. This seems to not be widely known.

8. Conclusion

We demonstrated a viable approach to detect three popular honeypots and one popular framework to create SMB honeypots (*Impacket*) on the Internet. Our validation has shown that exact fingerprint matches allow identification of a host with high accuracy.

We reused existing concepts such as the use of erroneous requests and the method of a most distinctive probe to separate honeypots from regular implementations. However, our results indicate that TLS fingerprints

5. https://github.com/wallix/redemption

are well suited to be combined with a honeypot detector. The results of our experiment on self-hosted honeypots suggest that attackers also employ fingerprinting methods to avoid both low- and high-interaction honeypots.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. We would like to thank Visuality Systems for providing us with a free license of their product YNQ and for the inspiring discussions about the SMB protocol. This work was partially funded by the German Federal Ministry of Education and Research under the project PRIMEnet, grant 16KIS1370.

References

[1] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. IEEE, 2005, pp. 29–36.

[2] E. López-Morales, C. Rubio-Medrano, A. Doupé, Y. Shoshitaishvili, R. Wang, T. Bao, and G.-J. Ahn, "Honeyplc: A next-generation honeypot for industrial control systems," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 279–291.

[3] S. Shan, E. Wenger, B. Wang, B. Li, H. Zheng, and B. Y. Zhao, "Gotta catch'em all: Using honeypots to catch adversarial attacks on neural networks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 67–83.

[4] Shodan.io, "Honeyscore," https://honeyscore.shodan.io/.

[5] ShodanHQ, "Honeyscore announcement," https://twitter.com/shodanhq/status/1311661444765806593, 10 2020, Twitter.

[6] R. Blog, "Nicer protocol deep dive: Internet exposure of remote desktop (rdp)," https://blog.rapid7.com/2020/10/23/nicer-protocol-deep-dive-internet-exposure-of-remote-desktop-rdp/, 2020.

[7] S. Morishita, T. Hoizumi, W. Ueno, R. Tanabe, C. Gañán, M. J. van Eeten, K. Yoshioka, and T. Matsumoto, "Detect me if you oh wait. an internet-wide view of self-revealing honeypots," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 134–143.

[8] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[9] D. Sysman, G. Evron, and I. Sher, "Breaking honeypots for fun and profit," *Black hat USA*, 2015.

[10] ——, "Breaking honeypots for fun and profit," *32c3 - Chaos Communication Congress*, 2015.

[11] J. Uitto, S. Rauti, S. Laurén, and V. Leppänen, "A survey on anti-honeypot and anti-introspection methods," in *World Conference on Information Systems and Technologies*. Springer, 2017, pp. 125–134.

[12] N. Krawetz, "Anti-honeypot technology," *IEEE Security & Privacy*, vol. 2, no. 1, pp. 76–79, 2004.

[13] X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham, "On recognizing virtual honeypots and countermeasures," in *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*. IEEE, 2006, pp. 211–218.

[14] S. Mukkamala, K. Yendrapalli, R. Basnet, M. Shankarapani, and A. Sung, "Detection of virtual environments and low interaction honeypots," in *2007 IEEE SMC Information Assurance and Security Workshop*. IEEE, 2007, pp. 92–98.

[15] L. Metongnon and R. Sadre, "Beyond telnet: Prevalence of iot protocols in telescope and honeypot measurements," in *Proceedings of the 2018 workshop on traffic measurements for cybersecurity*, 2018, pp. 21–26.

[16] V. Ghiëtte, H. Griffioen, and C. Doerr, "Fingerprinting tooling used for {SSH} compromisation attempts," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 61–71.

[17] J. van Eesteren, "Remote Desktop Protocol (RDP) 10 AVC/H.264 improvements in Windows 10 and Windows Server 2016 Technical Preview," https://techcommunity.microsoft.com/t5/microsoft-security-and/remote-desktop-protocol-rdp-10-avc-h-264-improvements-in-windows/ba-p/249588, 2016.

[18] *Remote Desktop Protocol: Basic Connectivity and Graphics Remoting*, Microsoft, 8 2020, Revision 53.0.

[19] GitHub, "Heralding," https://github.com/johnnykv/heralding, 2020.

[20] ——, "Rdpy," https://github.com/citronneur/rdpy, 2020.

[21] ——, "Dionaea," https://github.com/DinoTools/dionaea, 2020.

[22] SecureAuth, "Impacket," https://www.secureauth.com/labs/open-source-tools/impacket/, 2020.

[23] M. Nawrocki, M. Wählisch, T. C. Schmidt, C. Keil, and J. Schönfelder, "A survey on honeypot software and data analysis," *arXiv preprint arXiv:1608.06249*, 2016.

[24] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of Age: A Longitudinal Study of TLS Deployment," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 415428.

[25] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The Security Impact of HTTPS Interception," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/security-impact-https-interception/

[26] J. Althouse. TLS Fingerprinting with JA3 and JA3S. https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967.

[27] Cisco. Mercury: network fingerprinting and packet metadata capture. https://github.com/cisco/mercury.

[28] ——. Joy. https://github.com/cisco/joy.

[29] J. Althouse. Easily Identify Malicious Servers on the Internet with JARM. https://engineering.salesforce.com/easily-identify-malicious-servers-on-the-internet-with-jarm-e095edac525a?gi=3d3703067810.

[30] D. Dittrich, E. Kenneally *et al.*, "The menlo report: Ethical principles guiding information and communication technology research," *US Department of Homeland Security*, 2012.

[31] C. Partridge and M. Allman, "Addressing Ethical Considerations in Network Measurement Papers," *Communications of the ACM*, vol. 59, no. 10, Oct. 2016.

[32] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast internet-wide scanning and its security applications," in *USENIXSEC*, Washington, D.C., USA, 2013.

[33] GoSecure, "Pyrdp," https://github.com/GoSecure/pyrdp, 2018.

[34] G. Wan, L. Izhikevich, D. Adrian, K. Yoshioka, R. Holz, C. Rossow, and Z. Durumeric, "On the Origin of Scanning: The Impact of Location on Internet-Wide Scans," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20. New York, NY, USA: Association for Computing Machinery, 2020.

Appendix

TABLE 5. TOP 50 AUTONOMOUS SYSTEMS HOSTING HONEYPOTS

CO	ASN	Organization	SMB	RDP	**Total**
US	16509	AMAZON	232	167	399
US	20473	CHOOPA	126	95	221
US	14061	DIGITALOCEAN	102	90	192
DE	197540	netcup	66	72	138
TW	1659	TANet	131	1	132
US	8075	MICROSOFT	48	25	73
US	63949	Linode	33	37	70
US	14618	AMAZON	41	28	69
US	15169	GOOGLE	35	32	67
US	22773	Cox Communications	50	3	53
HK	135377	UCLOUD	26	25	51
FR	16276	OVH	17	22	39
CA	32613	IWEB	2	34	36
CN	132203	TENCENT	9	26	35
CA	25820	IT7NET	30	0	30
JP	2500	WIDE-BB WIDE Project	30	0	30
CN	45102	Alibaba	20	9	29
IT	137	GARR	12	16	28
JP	2497	Internet Initiative Japan	25	1	26
GB	9009	M247	12	10	22
DE	3320	DTAG	9	13	22
LT	56630	MELBICOM	8	10	18
JP	9370	SAKURA Internet Inc.	9	7	16
HK	136907	HUAWEI CLOUDS	9	3	12
US	7922	COMCAST	0	11	11
JP	2514	NTT PC Communications	6	5	11
JP	4713	NTT PC Communications	6	4	10
CN	45062	NETEASE	0	10	10
FR	12876	Online SAS	1	8	9
CZ	5588	GTS Central Europe	6	3	9
IT	3269	Telecom Italia	2	7	9
IN	4755	TATA Communications	4	5	9
MY	38182	Extreme Broadband	9	0	9
US	46562	PERFORMIVE	2	6	8
DE	51167	CONTABO	3	5	8
CA	31798	DATACITY	0	8	8
US	63473	HOSTHATCH	4	4	8
CA	136258	OBrainStorm Network Inc	4	4	8
EE	206804	ESTNOC	2	6	8
AU	133159	Mammoth Media Pty Ltd	4	4	8
RS	8400	TELEKOM SRBIJA	7	0	7
AR	263812	IPXON Networks	3	4	7
CN	4134	CHINANET	0	7	7
IR	58224	PJS	7	0	7
ES	39020	COMVIVE	3	3	6
GR	6799	OTENET	3	3	6
NL	6830	Liberty Global	2	4	6
JP	9355	NICT	1	5	6
HK	137280	Kingsoft cloud corporation	3	3	6
US	36352	COLOCROSSING	3	3	6

TABLE 6. RESPONSE COMPARISON FOR **PROTOCOL_RDP**

Field name	**XRDP**	**Win10**	**Win8**	**Win7**	**WinXP**	***RDPY***	***Heralding***
T.125 Conn. Resp.							
...							
Domain Parameters							
Max Channel IDs	22	✖	34	34	✖	22	✖
...							
RDP Server Data							
Server Core Data							
...							
Length	12	✖	16	12	✖	16	✖
Early Capability Fl.	✖	✖	1	✖	✖	0	✖
...							

TABLE 7. MAPPING BETWEEN WINDOWS END-USER AND SERVER VERSIONS

End-user	Server
XP	2003
7	2008R2
8	2012R2
10	2016, 2019

FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption

Fabian Franzen
Technical University of Munich
franzen@sec.in.tum.de

Manuel Andreas
Technical University of Munich
manuel.andreas@tum.de

Manuel Huber
Fraunhofer AISEC
manuel.huber@aisec.fraunhofer.de

ABSTRACT

(Dataset/Tool Paper) To secure mobile devices, such as laptops and smartphones, against unauthorized physical data access, employing Full Disk Encryption (FDE) is a popular defense. This technique is effective if the device is always shut down when unattended. However, devices are often suspended instead of switched off. This leaves confidential data such as the FDE key, passphrases and user data in RAM which may be read out using cold boot, JTAG or DMA attacks. These attacks can be mitigated by encrypting the main memory during suspend. While this approach seems promising, it is not implemented on Windows or Linux.

We present FridgeLock to add memory encryption on suspend to Linux. Our implementation as a Linux Kernel Module (LKM) does not require an admin to recompile the kernel. Using Dynamic Kernel Module Support (DKMS) allows for easy and fast deployment on existing Linux systems, where the distribution provides a prepackaged kernel and kernel updates. We tested our module on a range of 4.19 to 5.3 kernels and experienced a low performance impact, sustaining the system's usability. We hope that our tool leads to a more detailed evaluation of memory encryption in real world usage scenarios.

ACM Reference Format:
Fabian Franzen, Manuel Andreas, and Manuel Huber. 2024. FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption. In *Proceedings of CODASPY '20*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our society and businesses rely on the availability of secure computing devices such as notebooks or desktop PCs. As data theft may result in disclosure of important business secrets or sensitive personal information, these devices need to be protected from unauthorized access. A threat special to portable devices is that they can be easily stolen once unattended.

To counter the risk of data theft, many businesses and individuals rely on FDE to protect sensitive data on their devices. Without the appropriate passphrase or smart card, attackers will not be able to extract sensitive information from a switched off device. Depending on the Operating System (OS), FDE can be employed e.g. using Microsoft Bitlocker, Linux dm-crypt or Apple FileVault. Considering smartphones, both iOS and Android have also integrated FDE[7].

As a plus, iOS minimizes the presence of sensitive material in RAM using encryption aware storage controllers with secure RAM for key material. This reduces the attack surface to adversaries capable of obtaining a memory dump, but does not fully mitigate it as applications may still store sensitive information in RAM.

Common solutions do not protect this temporary data, such as passphrases or the FDE key, if the device is not fully switched off. We believe many users, not switching off their devices for faster wake up, are unaware of this attack surface. Attacks using physical properties of the hardware (e.g. cold-boot attacks [10]) or using unsecured peripheral connectors could be leveraged by an adversary to extract it. Such connectors could be JTAG, DMA-enabled ones (like Firewire [1] or Thunderbolt) or free memory DIMM sockets [21].

Unfortunately, hardware trust anchors like Trusted Platform Modules (TPMs) are usually only involved in the initial authentication process at boot. After successful authentication, the FDE key typically resides outside the TPM in normal unsecured RAM. IOS stores the FDE key solely on its Secure Enclave Processor[13], never exposing it to the main CPU. However, such special coprocessors are not available on off-the-shelf laptops or PCs.

Suspend-time memory encryption approaches, proposed in previous research, en-/decrypt memory during suspend and resume cycles. This has the advantage that once suspended, the FDE key no longer needs to be present on the system. As a result, even attackers with control over the CPU cannot read sensitive memory contents. Moreover, suspend-time encryption only impacts performance during suspension and resumption, but not during runtime. When pursuing the goal to protect unattended devices, suspend-time encryption represents an efficient and secure approach if specialized hardware is not available. Unfortunately, previous research only provided kernel patches for Linux, which have quickly become outdated as the kernel evolves. Therefore, we still lack an easy-to-use implementation to prove that this concept works in real-world setups. In summary, we make the following contributions:

- We provide FridgeLock, a tool to study the impact of suspend time memory encryption on real world setups.
- FridgeLock is designed as an LKM, such that suspend time memory encryption can easily be tested on a large number of Linux distributions without the need to recompile their kernel. We achieve this using DKMS to recompile the LKM in case of security updates. This results in a solution more agnostic to kernel changes.
- We tested our module on various distributions on the x86 platform and provide performance measurements, showing a user-acceptable performance for real world usage.

CODASPY '20, March 16–18, 2020, New Orleans, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

Several lines of work for protecting sensitive data in main memory exist. Key hiding techniques [6, 9, 16, 19] offload specific keys, such as the FDE key, from RAM to CPU or GPU registers and caches and execute the cipher exclusively on-chip. Besides being platform-dependent and hardly portable, key hiding mechanisms usually have a strong impact on performance. These mechanisms only protect a specific key and its associated cipher from memory attacks, but leave all other sensitive parts of main memory unprotected. Further, attackers gaining privileges on the system can directly access the keys, whereas with our approach, the memory encryption key is not available when the system is suspended.

Hardware-assisted memory encryption architectures [4, 20, 23] transparently encrypt main memory throughout the whole runtime of the system, keeping the encryption key and cipher computation off the main application processor. However, these architectures are not available on end user systems. Architectures designed by major hardware vendors, like AMD Secure Encrypted Virtualization (SEV) or Intel Multi-Key Total Memory Encryption (MKTME), target server systems only. Their goal is to protect memory against physical attackers and, at least in case of SEV, from a malicious or compromised hypervisor. SEV, for instance, stores the encryption key in an isolated coprocessor but has been demonstrated to be vulnerable against various side-channel attacks [14, 15, 22] allowing to extract encrypted memory in plaintext or to obtain privileges on encrypted guests. This shows that the reliable protection of main memory at all times poses a hard challenge.

Processors for consumer devices rather offer extensions to provide secure enclaves, such as ARM TrustZone, or Intel Software Guard Extensions (SGX). Enclaves enable shielded execution of sensitive code and the storage of data secure from both memory attacks and compromised OSs. These extensions can be leveraged as building blocks for security architectures, for instance, for use as isolated environments for encryption key storage and cipher execution [11].

Runtime memory encryption has also been realized in software. Some approaches leave a portion of memory unencrypted in a sliding window while most of the main memory is encrypted [5, 8, 17, 18]. In general, software-based runtime encryption approaches come with a notable impact on performance and can not provide memory protection against attackers gaining privileges on the system.

We base our memory encryption approach on *Freeze & Crypt* [12]. Further suspend-time approaches for x86 platforms are *Transient Authentication* [3] and *Hypnoguard* [24]. *Transient Authentication* encrypts main memory using a hardware token that provides the encryption keys [2]. When the token is removed, user space processes get suspended and their memory encrypted. Because suspension and resumption took about eight seconds, an application-aware mode was proposed. This allowed the protection of only specific assets using an API, which requires modifying applications.

Hypnoguard [24] hooks en-/decryption of memory into phases of suspension and resumption where the OS is no longer, respectively not yet, active. This allows to encrypt the whole memory without considering process mappings and without requiring kernel support but has the disadvantage that the kernel's support for hardware devices (such as displays, keyboards) is not available. The design requires implementing custom hardware-specific crypto routines and drivers to interact with hardware devices, such as for passphrase input. A TPM is used to protect the encryption while the cipher is executed in Intel's Trusted Execution Technology (TXT) environment. These design decisions make *Hypnoguard* less applicable in practice, while we require only adding an LKM to a system.

3 DESIGN

Like most of the previous approaches, FridgeLock targets Linux, because of the availability of source code. Further, we believe that memory encryption could be ported to other OSs if we can show feasibility on one of them. In this section, we derive the design of FridgeLock, based on the following design goals:

Easy Integration All existing academic approaches we are aware of are implemented as kernel patches, which forces the end user to recompile their kernel if they want to protect their system and, additionally, on every kernel update. To allow for a wide potential user base and to ease kernel updates with prepackaged distribution updates, we developed FridgeLock as an LKM. This allows distribution of FridgeLock as a binary or through dynamic compilation on every kernel update using DKMS.
Ultimately, we hope that FridgeLock will be integrated into the Linux kernel.

Low Performance Overhead We seek to keep the impact on performance as low as possible to not adversely impact user experience. Our LKM only hooks into the suspend and wakeup procedures, where small additional delays should be acceptable.

Protection of Sensitive Data Sensitive user data should be protected from adversaries under our assumed attacker model.

3.1 Attacker Model

We assume that an unattended device is stolen from a user in suspended state. In this state, the device is inspected by the attacker. Afterwards, the attacker can bypass all software and hardware mitigations (e.g. SEV) *somehow*. We only consider attacks where the device is lost *once* i.e. a device can not be stolen and given back. This excludes (1) *evil-maid attacks* where the attacker could e.g. corrupt the system and wait for the unknowing user to return, and (2) attacks involving an evil hardware vendor, who attacks the system from the HW side before it is stolen. As our design is not based on hardware trust anchors, we do not necessarily assume that TPMs or Secure Enclaves on the CPU are correctly implemented. An attacker may be able to execute arbitrary code (e.g. injected via JTAG or DMA) at any privilege level *after* the device is stolen.

Furthermore, our used cryptographic primitives (AES) and its operating mode (AES-XTS) have to be correctly implemented to be effective. This attacker model should be reasonable, given that e.g. TRESOR [16] was broken under similar assumptions [1].

3.2 Confidential Data in Memory

Given the goals and the attacker model, we evaluated the assets in the Linux kernel that need protection:

(A0) Filesystem. The files on disk including sensitive user and system owned files. These files are protected by FDE.
(A1) Page Cache. Opened files from disk may be cached in RAM in the page cache. As these are basically (partial) copies of files on disk, these can contain sensitive data.
(A2) Filesystem Metadata. Metadata of the filesystem such as filenames, modification times and folder structure. Less critical than actual file contents, but nonetheless sensitive.
(A3) Userspace memory. The active Virtual Memory Areas (VMAs) of running processes. May contain private keys, passphrases and (partial) copies of data from disk.
(A4) Free'd Pages. The Linux kernel does not clear pages that have just been released by the kernel or by a userspace process (e.g. explicitly or process termination).
(A5) Kernel Objects. For example, `task_struct`, i.e., internal information of running processes. It contains a CPU register snapshot, the process name, and a kernel stack reference.
(A6) Linux Keyring. The Linux keyring is a central key storage inside the kernel. More specifically the CIFS filesystem uses the keyring to store passphrases to accessed shares.
(A7) Disk encryption keys. The dm-crypt module is responsible for FDE and stores its keys outside of the Linux keyring.
(A8) Arbitrary Device Buffers. Any hardware might store sensitive I/O-data such as keystrokes.

In contrast to these assets, e.g. the *Linux Text Segment* or the *Firmware and Bootloader Code* do not require protection under our chosen attacker model as we exclude *evil maid attacks*.

3.3 Integration into Linux Power Management

In order to put the system into the S3 sleep state (Suspend-To-Ram), the Linux kernel first suspends execution of userspace processes (called *freezing*). This is necessary to stop processes from interfering with the suspend process.[1] Kernel threads are not *frozen* by default, but *freezable* kernel threads (e.g. threads that could cause the suspend process to fail) are stopped directly after the processes. Finally, the kernel notifies device drivers to put their device into a power saving sleep state. When this process is finished, execution on the CPU is halted until an interrupt initiates the resume, where this procedure is done vice versa.

We split FridgeLock into two parts: An LKM and a non-encrypted *helper process* running in userspace. The LKM is responsible for process memory encryption, for protecting the other assets, and for spawning the helper process right before suspension. The helper process is responsible for memory and disk encryption key management during wakeup cycles. For this purpose, the userspace process queries the current user for the decryption passphrase after system wakeup.

The LKM integrates with Linux power management using the *device power management* subsystem (through `register_pm_notifier()`) and through the device driver power management API (i.e. `dev_pm_ops`). In order to get access to this API, we register a virtual device together with a driver FridgeLock provides. Combined, this offers the following hooking points crucial to our design:

[1] see `Documentation/power/freezing-of-tasks.txt` in the kernel sources for further information.

(1) On Early Suspend: Before the system is going to freeze the processes.
(2) On Late Suspend: After the system has frozen the processes.
(3) On Resume: Before the system is going to thaw the processes.

In the following, we describe FridgeLock's tasks from initialization to suspension and resumption:

Initialization Time. At its initialization, the LKM creates a character device for ioctl-based communication with the helper process to: (1) force the helper process to sleep until system resume, (2) to send the read-in FDE passphrase to our LKM after entry and (3) to probe for encrypted partitions needing protection. Moreover, it hooks into the *device mapper* infrastructure to obtain the FDE key on set, which we make use of to encrypt the userspace memory.
Hook 1. In case of a system suspend, the LKM is notified through hooking point (1) and starts the helper process. Moreover, it sets a bit on the helper process which causes the regular *freezer* to skip this process.
Hook 2. At hooking point (2), when all other userspace processes are *frozen*, we encrypt the memory map of the userspace processes (except for special mappings). Further, we evict filesystem caches and overwrite unused pages with zeros afterwards. As a last step, we wipe out the FDE key. For the actual encryption operations, we utilize the kernel crypto API. This allows using hardware crypto accelerators, such as AES-NI, resulting in extremely fast encryption speeds. We encrypt each page individually and utilize AES-XTS as cipher mode as our encryption requirements are almost identical to typical FDE, for which AES-XTS is recommended by NIST [**?**]. We use the physical page addresses as IVs to guarantee unique IVs for every page. To wipe and restore the FDE key, FridgeLock relies on the *suspend*, *resume* and *message* operations of the *dm-crypt* module.
Hook 3. On resume (3), the LKM wakes the helper process. The helper, in turn, asks the user for the FDE passphrase and restores the prior state of all dm-crypt devices using the regular *cryptsetup* toolchain. The LKM then decrypts userspace memory before the system returns to normal operation.

4 IMPLEMENTATION

In the following, we describe the implementation of our FridgeLock prototype, first our userspace helper followed by the LKM.

Userspace Helper. We partitioned our *userspace helper* into two parts, which we call *Stage 1* and *Stage 2*. Figure 1 reflects that the LKM spawns the *Stage 1* helper process at hooking point (1). The *Stage 1* process discovers the dm-crypt volumes on the system that need to be suspended, see Figure 1. Furthermore, it sets up an initramfs like tmpfs containing the *Stage 2* helper process and its necessary runtime environment, e.g. libcryptsetup. The *Stage 1* process then chroots into this new environment and executes the *Stage 2* process inside the chroot. This chroot operation is necessary because the rootfs is not available between wakeup and passphrase entry of the user. The *Stage 2* process is responsible for signaling the discovered volumes to the LKM via an ioctl, which will suspend and wipe the keys at hooking point (2). Further, the process asks the user for the passphrase for resumption (see hooking point 3,

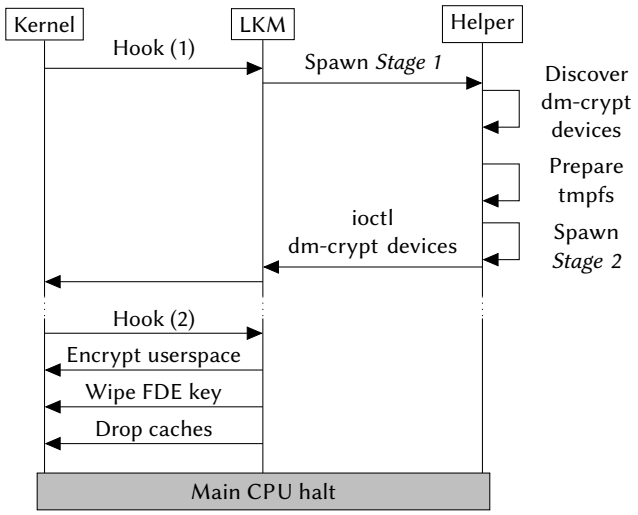

Figure 1: Sequence diagram of the suspend procedure.

omitted from Figure 1). On startup, the discovered volumes are received as parameters from the *Stage 1* process and transferred to the LKM via an ioctl. The LKM, in turn, suspends execution of the *Stage 2* process and returns control to the kernel.

Kernel Module. After our LKM gets notified of completed process freezing, see hooking point (2) in Figure 1, it iterates through the dm-crypt volumes received by the *Stage 1* process and suspends them using the device mapper kernel component. This suspend operation ensures that the encryption key is securely wiped from RAM and makes any further IO operation on the affected volume to be blocking until resumption. Before suspension of the *dm-crypt volumes*, it will place a kprobes based hook to extract the FDE key, which we use to encrypt userspace. The projects *arch-luks-suspend*[2] and its successor *go-luks-suspend*[3] provide an implementation to remove the FDE key during suspension. We utilized parts of their implementation to realize our *Stage 2* process.

Afterwards, the virtual address space encryption of all userspace processes takes place as follows. We start by iterating through all VMAs of our helper process. To avoid their encryption, we mark all pages belonging to those VMAs as "already encrypted" by setting a flag on the kernel `struct page`. In the next step, we iterate through all frozen userspace and kernel tasks, encrypt their VMAs and thus their pages in-place. Before carrying out the encryption of a page, we first check if the "already encrypted" flag is set, then if the page belongs to a special region (e.g. DMA or device memory), and if not encrypt it and set the flag. This approach ensures that neither the VMAs of our helper process nor mapped hardware memory, unaware of our encryption, are encrypted. On resume, we use our kprobes hook into the device mapper to observe the FDE key set of the helper process and re-obtain it in the LKM.

Certain functionality of our LKM requires usage of unexported kernel functions. In order to call these functions anyways, we utilized the exported kernel function `kallsyms_lookup_name`, which is able to resolve a symbol's name to its address in memory. We resolve all unexported functions at module initialization and are thus able to call them at any point through function pointers. If no specialized hooking mechanism is provided by the kernel, we use *kprobes* and *kretprobes* to intercept function calls.

We utilized the `/proc/sys/vm/drop_caches` mechanism to clear the page cache. Using this mechanism, the kernel can be advised to drop page cache, dentries and inodes from memory. We instrumented the `invalidate_page_cache` function, which is part of the inode clearing procedure of the kernel, using the kprobes framework. This instrumentation simply zeros out pages belonging to inodes that are going to be erased.

[2]https://github.com/vianney/arch-luks-suspend
[3]https://github.com/guns/go-luks-suspend

5 EVALUATION

5.1 Completeness of Protection

Our implementation addresses (A0), (A1), (A3), (A4), (A7) by construction. Asset (A2) is partially protected as dropping the page cache also contains inode and dentry structs. However, this does not include all filesystem metadata. Moreover, the Linux keyring (A6) and arbitrary hardware buffers (A8) are not sufficiently protected. In the case of (A6), an API to stop kernel threads from accessing the invalid keys that a key-wipe would leave behind is missing. In the case of (A8), various drivers store their buffers at arbitrary locations (e.g. in the device memory itself, the kernel heap or kernel stack). Protecting these buffers would require knowing every drivers exact implementation.

Furthermore, we experimentally verified our approach by comparing a QEMU snapshot of an un- and FridgeLock-protected virtual machine. In both cases, a test program loads known confidential data from disk and places them in memory. Moreover, several applications like Firefox are started. While a scan of the unprotected dump did reveal the FDE key and multiple copies of the confidential test data, the protected instance did not. Additionally, we analyzed the snapshots for secrets of the remaining applications using *AESKeyFinder* (proposed by [10]), which searches for expanded AES keys. Our results showed that the unprotected snapshot reveals several keys, while our protected snapshot does not.

5.2 Performance

We tested FridgeLock on a Dell XPS 15 9550 with an Intel i7 6700HQ (2.6 GHz, 4 cores), 16 GB RAM (DDR4 2133MHz) and a PM951 Samsung 512GB SSD. The performance of FridgeLock is linearly dependent on the memory usage of the running processes, i.e., on the amount of memory to en-/decrypt.

Therefore, we constructed three scenarios to test FridgeLock: ① A minimal scenario with only a basic set of processes (init + bash without xserver, userspace footprint 77MB), ② an average load scenario (Gnome + a few Firefox instances, userspace footprint 5,5GB) and ③ a high load scenario where all RAM is occupied by userspace. The results are visualized in Figure 2. On system resume, about the same time span is needed for decryption of the processes. The overall decryption time is dominated by passphrase entry; the cryptographic operations take the same time as for encryption. Additionally, no time is spent on clearing caches during resume.

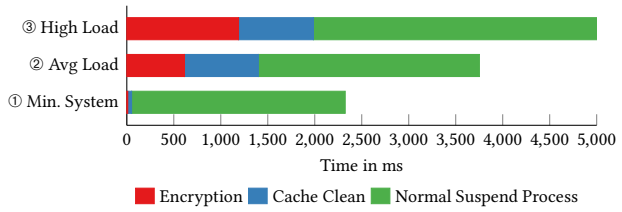

Figure 2: Overall measured system suspend time.

5.3 Maintainability

To implement our LKM, we were forced to load unexported functions via `kallsyms_lookup_name` and place `kprobes` hooks where necessary. These workarounds are generally discouraged as they are likely to break across different kernel releases, but are currently a necessity due to the lacking kernel API. While developing FridgeLock, we experienced both the kernel API and the unexported functions to undergo breaking changes, leading us to believe frequent maintenance of our LKM is necessary. We want to emphasize that this issue is just as present in a kernel patch implementation with the difference being that our LKM would possibly erroneously succeed building if an unexported function changed functionality or signature due to the nature of loading them via `kallsyms_lookup_name`. However, this could be avoided by additional automatic validation outside of compilation.

6 CONCLUSION

We presented FridgeLock, a tool for suspend time memory encryption on Linux. FridgeLock enables the protection of important assets in userspace and kernel memory on a suspended machine with deployed FDE. We successfully tested FridgeLock on x86 systems with kernel versions 4.19 to 5.3. Our evaluation on a mid-end notebook with 16GB RAM shows that FridgeLock's performance overhead is sufficiently small for use in practice, even in worst-case scenarios. This performance overhead is even more negligible considering it only affects suspend and resume operations.

Furthermore, our implementation as an LKM with userspace components results in an effortless installation and maintenance process for the end user through packaging and DKMS support. As usability is usually in direct conflict with security we deem the high usability of FridgeLock to be its strongest point.

Nonetheless, FridgeLock is currently not able to protect all sensitive assets in the kernel. First, the LKM design decision limits access to kernel internal structures. Even if the location in memory is known, we can not easily wipe information as we may not know all places where it is accessed in advance. Second, device drivers may contain numerous buffers with I/O data containing sensitive information through which we can not easily iterate. For instance, the keyboard driver may still contain the last typed passphrases before suspend. We consider the extension of the FridgeLock tool to locate and protect these buffers to be future work.

AVAILABILITY

To encourage open research, we open sourced our work at GitHub: https://github.com/fridgelock-lkm/fridgelock.

REFERENCES

[1] Erik-Oliver Blass and William Robertson. 2012. TRESOR-HUNT: attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 71–78.
[2] Mark D. Corner and Brian D. Noble. 2002. Zero-interaction Authentication. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking* (Atlanta, Georgia, USA) *(MobiCom '02)*. ACM, 1–11.
[3] Mark D. Corner and Brian D. Noble. 2003. Protecting Applications with Transient Authentication. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys '03)*. ACM, 57–70.
[4] Guillaume Duc and Ronan Keryell. 2006. CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, 483–492.
[5] Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller. 2016. HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. In *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 79–87.
[6] Johannes Götzfried and Tilo Müller. 2013. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *Proceedings of the 2013 International Conference on Availability, Reliability and Security (ARES '13)*. IEEE Computer Society, Washington, DC, USA, 161–168. https://doi.org/10.1109/ARES.2013.23
[7] Johannes Götzfried and Tilo Müller. 2014. Analysing Android's Full Disk Encryption Feature. *JoWUA* 5, 1 (2014), 84–100.
[8] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. 2016. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. ACM, 919–924.
[9] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with Private Keys without RAM. In *NDSS*. 23–26.
[10] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.
[11] Julian Horsch, Manuel Huber, and Sascha Wessel. 2017. TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor. In *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications* (Sydney, Australia) *(TrustCom '17)*. IEEE, 152–161.
[12] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. 2018. Freeze and Crypt: Linux Kernel Support for Main Memory Encryption. *Computers & Security* 86 (2018), 420 – 436. https://doi.org/10.1016/j.cose.2018.08.011
[13] Apple Inc. 2019. iOS Security - iOS 12.3. https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf
[14] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association.
[15] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec'18)*. ACM.
[] Tilo Müller, Felix C. Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Conference on Security* (San Francisco, CA) *(SEC'11)*. USENIX Association, Berkeley, CA, USA, 17–17. http://dl.acm.org/citation.cfm?id=2028067.2028084
[16] Tilo Müller, Felix C Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM.. In *USENIX Security Symposium*, Vol. 17.
[17] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. 2017. No Sugar but All the Taste! Memory Encryption Without Architectural Support. In *Computer Security – ESORICS 2017*. 362–380.
[18] Peter A. H. Peterson. 2010. Cryptkeeper: Improving security with encrypted RAM. In *IEEE International Conference on Technologies for Homeland Security (HST)*. 120–126.
[19] Patrick Simmons. 2011. Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, 73–82.
[20] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. 2007. Aegis: A Single-Chip Secure Processor. In *IEEE Design & Test*, Vol. 24. IEEE Computer Society Press, 570–580.
[21] Anna Trikalinou and Dan Lake. 2017. Taking DMA attacks to the next level. *BlackHat USA* (2017).
[22] Jan Werner, Joshua Mason, and et al. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS 2019)*. 73–85.
[23] Alexander Würstlein, Michael Gernoth, Johannes Götzfried, and Tilo Müller. 2016. Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure. In *Proceedings of the 29th International Conference on Architecture of Computing Systems (ARCS '16, Vol. 9637)*. 60–71.

[24] Lianying Zhao and Mohammad Mannan. 2016. Hypnoguard: Protecting Secrets Across Sleep-wake Cycles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 945–957.

Bibliography

Please note: This Bibliography does *not* include the references of the embedded research papers. The chosen numeric reference numbers may collide with the reference numbers used in the research papers.

[1] Google Threat Analysis Group, "Tool of first resort - israel-hamas war in cyber," Google, Tech. Rep., 2024.

[2] Google Threat Analysis Group, "Fog of war: How the ukraine conflict transformed the cyber threat landscape," Google, Tech. Rep., 2023.

[3] Bundesministerium für Sicherheit in der Informationstechnik (BSI), "Die Lage der IT-Sicherheit in Deutschland 2023," 2023.

[4] Statista. "Market share of mobile operating systems worldwide from 2009 to 2023, by quarter." (2024), [Online]. Available: `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`.

[5] Microsoft. "Linux virtual machines in azure." (2024), [Online]. Available: `https://azure.microsoft.com/en-us/products/virtual-machines/linux`.

[6] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[7] The Volatility Foundation. "What is the volatility framework?" (2024), [Online]. Available: `https://volatilityfoundation.org/the-volatility-framework/`.

[8] Google. "Rekall forensics github repo." (2020), [Online]. Available: `https://github.com/google/rekall?`.

[9] A. Tanenbaum, *Modern Operating Systems*. Pearson Education, Inc., 2009.

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[11] PAX Team. "Address Space Layout Randomization (ASLR)." (2002), [Online]. Available: `https://pax.grsecurity.net/docs/aslr.txt`.

[12] Grsecurity. "KASLR: An exercise in cargo cult security." (2013), [Online]. Available: `https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security`.

[13] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000# ifdefs issue," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 421–432.

[14] Linux Kernel Mailing List. "The kernel test robot auditing patch cc5dc7e7f93009390baa8587837741bde (2024), [Online]. Available: `https://lore.kernel.org/linux-arm-kernel/202404301818.b59E8LWv-lkp@intel.com/`.

[15] L. Spitzner, "Honeypots: Tracking hackers," 2002.

[16] T. H. Project. "The honeynet project - about us." (2024), [Online]. Available: `https://www.honeynet.org/about/`.

[17] Y.-M. Wang, D. Beck, X. Jiang, *et al.*, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Network and Distributed System Security Symposium*, 2006.

[18] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. C. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *International Symposium on Recent Advances in Intrusion Detection*, 2006.

[19] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, "Container orchestration honeypot: Observing attacks in the wild," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23, , Hong Kong, China, Association for Computing Machinery, 2023, pp. 381–396, ISBN: 9798400707650. DOI: 10.1145/3607199.3607205.

[20] N. Provos, "A virtual honeypot framework," in *USENIX Security Symposium*, 2004.

[21] E. López-Morales, C. E. Rubio-Medrano, A. Doupé, *et al.*, "Honeyplc: A next-generation honeypot for industrial control systems," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[22] S. Srinivasa, J. M. Pedersen, and E. Vasilomanolakis, "Interaction matters: A comprehensive analysis and a dataset of hybrid iot/ot honeypots," *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[23] I. Livshitz. "What's the difference between a high interaction honeypot and a low interaction honeypot?" (2019), [Online]. Available: `https://www.akamai.com/blog/security/high-interaction-honeypot-versus-low-interaction-honeypot-comparison`.

[24] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[25] T. Garfinkel, M. Rosenblum, *et al.*, "A virtual machine introspection based architecture for intrusion detection.," in *Ndss*, San Diega, CA, vol. 3, 2003, pp. 191–206.

[26] Open Source. “Libvmi: Simplified virtual machine introspection.” (2015), [Online]. Available: https://libvmi.com/.

[27] Open Source. “Xenaccess library.” (2009), [Online]. Available: https://code.google.com/archive/p/xenaccess/.

[28] B. D. Payne, D. d. A. Martim, and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE, 2007, pp. 385–397.

[29] Open Source. “Panda, an open-source platform for architecture-neutral dynamic analysis.” (2024), [Online]. Available: https://panda.re/.

[30] Open Source. “Drakvuf - black-box binary analysis system.” (2024), [Online]. Available: https://drakvuf.com/.

[31] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[32] S. Proskurin, J. Kirsch, and A. Zarras, “Follow the whiterabbit: Towards consolidation of on-the-fly virtualization and virtual machine introspection,” in *ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings 33*, Springer, 2018, pp. 263–277.

[33] N. Stephens, J. Grosen, C. Salls, *et al.*, “Driller: Augmenting fuzzing through selective symbolic execution,” in *Network and Distributed System Security Symposium*, 2016.

[34] angr Project contributors. “Angr documentation.” (2024), [Online]. Available: https://docs.angr.io/en/latest/index.html#.

[35] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems,” in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, IEEE, 1997, pp. 67–72.

[36] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “Sok: Automated software diversity,” in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 276–291.

[37] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, “Compiler-assisted code randomization,” in *2018 IEEE symposium on security and privacy (SP)*, IEEE, 2018, pp. 461–477.

[38] Linux Kernel Mailinglist. “Function granular kernel address space layout randomization (fg-kaslr).” (2020), [Online]. Available: https://lore.kernel.org/lkml/20220209185752.1226407-1-alexandr.lobakin@intel.com/.

[39] D. Williams-King, G. Gobieski, K. Williams-King, *et al.*, "Shuffler: Fast and deployable continuous code {re-randomization}," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 367–382.

[40] Z. Durumeric, E. Wustrow, and J. A. Halderman, "{Zmap}: Fast internet-wide scanning and its security applications," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 605–620.

[41] Electronic Frontier Foundation. "The eff ssl observatory." (2010), [Online]. Available: `https://www.eff.org/observatory`.

[42] D. Adrian, K. Bhargavan, Z. Durumeric, *et al.*, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 5–17.

[43] Google. "Oss-fuzz." (2024), [Online]. Available: `https://google.github.io/oss-fuzz/`.

[44] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[45] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[46] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "Diffuzz: Differential fuzzing for side-channel analysis," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 176–187, 2018.

[47] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 859–874.

[48] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1683–1700.

[49] O. A. V. Ravnås. "Frida - dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." (2024), [Online]. Available: `https://frida.re`.

[50] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[51] Github. "Lmbench." (2021), [Online]. Available: `https://github.com/intel/lmbench`.

[52] A. Oliveri, M. Dell'Amico, and D. 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.

[53] A. Oliveri, “A zero-knowledge approach to memory forensics,” Ph.D. dissertation, Sorbonne Université, 2023.

[54] D. Maier, F. Franzen, and M. Wagner, “Mehr schlecht als recht: Grauzone sicherheitsforschung: Reverse engineering vor gericht,” *Datenschutz und Datensicherheit-DuD*, vol. 44, no. 8, pp. 511–517, 2020.

[55] GitHub. “Selinux project.” (2024), [Online]. Available: `https://github.com/SELinuxProject`.

[56] AppArmor Author Group. “Apparmor - linux kernel security module.” (2024), [Online]. Available: `https://apparmor.net/`.

[57] eBPF.io authors. “Ebpf - dynamically program the kernel for efficient networking, observability, tracing, and security.” (2024), [Online]. Available: `https://ebpf.io/`.

[58] F. Lawler. “Live-patching security vulnerabilities inside the linux kernel with ebpf linux security module.” (2022), [Online]. Available: `https://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm`.

[59] Deutsche Telekom AG. “Sicherheitstacho - Über Uns.” (2024), [Online]. Available: `https://www.sicherheitstacho.eu/#/de/about`.

Licences and Permission Statements

This chapter contains the license/permission statements from the respective publishers that grant the author the right to use the publication in this thesis.

- The publications *RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis* and *Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots* are published below a Creative Commons Open Access licence, which can be accessed via `https://creativecommons.org/licenses/by-nc-sa/4.0/`.
- The IEEE permission statement for the publication *Looking for Honey Once Again: Detecting RDP and SMB Honeypots in the Internet* can be found on the following pages.
- The ACM licencing terms for the publication *FridgeLock: Preventing Data Theft on Suspended Linux with Usuable Memory Encryption* can be found on the following pages.

Looking for Honey Once Again: Detecting RDP and SMB Honeypots on the Internet

Conference Proceedings:
2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)

Author: Fabian Franzen

Publisher: IEEE

Date: June 2022

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

ACM Author Gateway

Author Resources

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

After an ACM journal submission has been accepted and has entered the production process, ACM makes the Author’s Accepted Manuscript (AAM) available for preview under the ACM “Just Accepted” program until the “Version of Record” is available and assigned to its proper issue. The AAM carries the article’s permanent DOI and can be cited immediately.

Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM