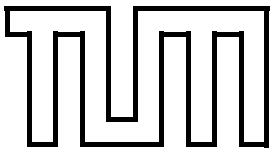


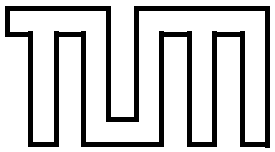
TUM School of Computation, Information and Technology
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

Virtualization-assisted Dynamic Binary Analysis and Operating System Security

Sergej Proskurin

Dissertation



TUM School of Computation, Information and Technology
der Technischen Universität München

Virtualization-assisted Dynamic Binary Analysis and Operating System Security

Sergej Proskurin

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Jens Grossklags, Ph.D.

Prüfer der Dissertation:
1. Prof. Dr. Claudia Eckert
2. Prof. Vasileios P. Kemerlis, Ph.D.
3. Prof. Michalis Polychronakis, Ph.D.

Die Dissertation wurde am 06.04.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.08.2022 angenommen.

Acknowledgements

I have stumbled across the fascinating world of virtualization technology very early in the second year of my studies. I remember enjoying my first presentation about microkernels and hypervisors. At the time, even though I was instantaneously intrigued by the call of the concepts, I did not expect this path to take me on such an adventurous journey, which has eventually led to this work. Along the way, I have stumbled across a number of great findings, insightful acquaintances, and inspiring collaborations, which, in the end, have helped me putting the pieces together. For this I am grateful.

This dissertation would not have been possible without constant support and encouragement of many people, to whom I would like to express my gratitude. First and foremost, I would like to thank Prof. Dr. Claudia Eckert for giving me the opportunity to pursue a doctoral degree in virtualization-assisted security and to fulfill my dream of spending time in the United States. Also, I would like to specially thank my supervisors and second advisors at the Stony Brook and Brown University, Prof. Dr. Michalis Polychronakis and Prof. Dr. Vasileios Kemerlis, for your incredible support, inspiration, and guidance. I have greatly enjoyed working with you and debating in our endless discussions. You have been great advisors and have helped me push through, even in the most frustrating hours.

Thank you Prof. Dr. Apostolis Zarras and Prof. Dr. Jens Grossklags. Over the years, both of you have given me a strong and persistent hold, encouragement, and assistance on a professional but also on a personal level. I feel lucky having both of you on my side.

My special gratitude to Dr. Tamas Lengyel, Dr. Sebastian Vogl, and Dr. Jonas Pfoh. Your work has inspired and helped me finding my own way into the world of virtualization.

A huge thanks to all students at the Technical University of Munich, who have assisted me over the years. In particular, I would like to express my gratitude to the virtualization team, comprising Marius Momeu, Christopher Roehmheld, Florian Jakobsmeier, Charlie Groh, and Ulrich Fourier. Without any doubt, you have been a helping hand.

Also, a huge thanks to my dearest colleagues at the Chair for IT Security as well as the Secure Operating Systems department at Fraunhofer AISEC. In particular, I would like to personally thank Steffen Wagner. It was my pleasure working with you.

I would like to thank my family and closest friends for the unconditional love, understanding, and support throughout my entire studies.

Finally, I would like to thank Marina, my dearest and best friend. I wholeheartedly thank you for your patience, your support on so many levels, your wisdom and great advice, your persistent encouragement and believe in my abilities, and for so many more super powers. You have been a rock, my strongest hold, throughout this incredible ride. Thank you.

Abstract

Virtualization technology has undergone a paradigm shift, in which it has turned its focus from virtualizing servers to assisting the security of operating systems. The technological drive has made virtualization an omnipresent and integral element in the digital world. Most of the modern application processors have added a wide-ranging set of hardware extensions to support system virtualization. The accompanied technological advances and the resulting benefits in regard to resource isolation have inspired researchers to change their perspective on virtualization and to form novel virtualization-assisted techniques for dynamic binary analysis and operating system defenses. Their motivation has helped them to repurpose the virtualization extensions for security. Specifically, they suggested to move security services out of the operating system into an isolated environment guarded by the hypervisor. Over time, virtual machine introspection have evolved and proven effective, even against sophisticated malicious actors; the strong isolation and inspection capabilities of the hypervisor ensure that even compromised virtual machines cannot easily mislead or directly manipulate the security frameworks, which retain an unimpaired view over the virtual machine's binary state. These capabilities have led to a substantial rise in adoption of security-driven virtualization techniques in private and industry sectors leveraging virtualization for *dynamic binary analysis* and *operating system security*.

Disregarding the benefits of virtualization-assisted security frameworks, prior work faces a set of challenges. For instance, a general limitation of virtualization-assisted security frameworks is that they cannot be deployed *on-demand*; they require a compliant hypervisor to be set up in advance underneath the target operating system to implement the necessary foundation. Further, modern virtual machine introspection based frameworks rely on hardware-supported capabilities to dynamically analyze sophisticated malware in a *stealthy* way. While these capabilities have evolved and shown great potential on x86-based, and especially Intel architectures, they lack the necessary foundation for virtual machine introspection to be equally effective on other architectures. For instance, this is one of the reasons why ARM has received insufficient attention with regard to stealthy analysis, despite its breakthrough and increasing acceptance in the server market. Undeniably, the motivation behind moving security services into isolated environments has helped security experts gain improved inspection, resilience, and stealth capabilities with

regard to virtual machine introspection based dynamic analysis of sophisticated malware. Yet, virtualization-assisted operating system security frameworks do not always need to fully outsource their logic and capabilities; this strategy constrains the portability of virtualization-assisted security measures as they rely on the logic integrated into the hypervisor or a security framework operating inside a different virtual machine.

In this work, we explore novel virtualization-assisted techniques to further enhance their capacity with regard to security by addressing the above challenges. Specifically, we position our research around two main pillars, which guide our work towards improving the state-of-the-art virtualization-assisted primitives for *dynamic binary analysis* and *operating system security*. We begin our work by establishing a foundation for both research directions that allows us to deploy arbitrary virtualization-assisted frameworks on-demand. In this context, we address the first of the presented challenges by introducing a thin microkernel-based hypervisor, WhiteRabbit, which we use to dynamically deploy virtualization-assisted primitives. We exemplify a scenario by installing a virtual machine introspection framework underneath a running Linux system, without leaving any in-guest artifacts behind. We demonstrate that the virtualization overhead of our prototype can compete with prominent hypervisors and hence presents a viable alternative.

We then turn our attention towards the first pillar, namely improving the state-of-the-art of virtualization-assisted primitives for dynamic binary analysis. Specifically, we develop novel primitives for setting and single-stepping software breakpoints on ARMv7 and ARMv8 application processors—specifically on the AArch32 and AArch64 execution state of the ARMv8 architecture—in a stealthy way. To achieve this, we repurpose the virtualization extensions to overcome the shortcomings of the hardware-intended monitoring mechanisms. We extend the Xen Project hypervisor with the ability to dynamically allocate and switch among different second level address translation tables on ARM to define different views on the guest-physical memory. To the best of our knowledge, we are the first to leverage this capability to hide software breakpoints in the guest's memory on AArch64. By additionally de-synchronizing the TLB-organization, we manage to establish a stealthy solution on AArch32. We demonstrate the effectiveness of our work by equipping the state-of-the-art dynamic binary analysis framework, DRAKVUF, with our primitives, and hence establish the foundation for stealthy dynamic binary analysis on ARM.

By addressing the second pillar of our research, we utilize virtualization to fortify security-critical operating system components. We introduce selective memory protection (xMP) primitives to empower guests with the ability to isolate and protect sensitive data in isolated xMP domains in kernel and user space against data-oriented attacks. We do no longer consider the system's virtualization extensions as components that are available solely to the hypervisor. Instead, we incorporate them as inherent building blocks into the operating system's subsystems, and hence allow them to define custom policies. Thus, we interface the Linux memory management with Xen to define different views on the guest-physical memory, which we repurpose for creating disjoint xMP domains. Combined with Intel's in-guest EPTP switching capabilities, we do not require the virtual machine to interact with the hypervisor to maintain the set of xMP domains, once they have been set

up. By additionally equipping pointers to data inside xMP domains with context-bound HMACs, we manage to obstruct data-oriented attacks. We apply xMP to process credentials and page tables in kernel space, and cryptographic material of selected applications in user space. We show that the xMP primitives entail only a limited overhead and present an effective defense against data-oriented attacks.

Finally, we conclude our research by focusing on the next higher abstraction level of the virtualization technology. Specifically, we enhance the security of modern operating system level virtualization techniques (containers) on Linux. We introduce Jesse, a static analysis based framework for tailoring policies for the Linux Secure Computing (seccomp) mode. Jesse utilizes an abstract interpretation based constant propagation to identify an over-approximated set of system call numbers the binaries in a specific Docker container are authorized to invoke. In addition, we combine Jesse with the state-of-the-art container and library debloating techniques to narrow down only the necessary regions of the `libc`, which Jesse considers during the analysis. Once we extract the authorized system calls, we compile seccomp policies to restrict access to unnecessary, and potentially vulnerable system calls. Through our prototype, we demonstrate that Jesse is an effective means to thwart existing real-world container escalation exploits.

Overall, through our research, we demonstrate that the potential of virtualization technology has not yet been explored to its full extent with regard to security. Our prototypes reveal that the introduced virtualization-assisted primitives do not only advance the state-of-the-art in dynamic binary analysis, but also open and prospect new horizons of novel operating system security architectures.

Kurzfassung

Die Virtualisierungstechnologie hat einen Paradigmenwechsel vollzogen, bei dem sie ihren Schwerpunkt von der Servervirtualisierung auf die Unterstützung der Sicherheit von Betriebssystemen verlagert hat. Der technologische Fortschritt hat die Virtualisierung zu einem allgegenwärtigen und integralen Bestandteil der digitalen Welt gemacht. Heute haben die meisten modernen Anwendungsprozessoren eine breit-gefächerte Palette von Hardwareerweiterungen zur Unterstützung der Virtualisierung eingeführt. Die sich daraus ergebenden Vorteile im Hinblick auf die Ressourcenisolierung haben Forscher dazu inspiriert, ihre Perspektive auf die Virtualisierungstechnologie zu ändern, um neue virtualisierungsgestützte Techniken für die dynamische Binäranalyse und den Schutz von Betriebssystemen zu entwickeln. Um dies zu erreichen, haben Forscher die Konzepte der hardwaregestützten Virtualisierungserweiterungen neu interpretiert und für die Systemsicherheit umfunktioniert. Insbesondere schlugen sie vor, Sicherheitsdienste aus dem Betriebssystem in eine isolierte Umgebung zu verlagern, die vom einem Hypervisor bewacht wird. Daraus hat sich im Laufe der Zeit ein neuer Mechanismus entwickelt, der als Introspektion virtueller Maschinen bezeichnet wird. Dieses Verfahren hat sich selbst gegen raffinierte böswillige Akteure als wirksam erwiesen. Die ausgeprägten Fähigkeiten des Hypervisors im Bezug auf die Ressourcenisolierung und Inspizierung stellen sicher, dass selbst kompromittierte virtuelle Maschinen die ausgelagerten Sicherheitsdienste nicht leicht irreführen oder direkt manipulieren können, weil die Sicherheitsdienste trotz einer potenziellen Systemkompromittierung eine unbeeinträchtigte Sicht auf den binären Zustand der virtuellen Maschine beibehalten. Aus diesem Grund haben diese Fähigkeiten zu einem erheblichen Anstieg der Akzeptanz von sicherheitsorientierten Virtualisierungstechniken im privaten sowie im industriellen Sektor für die *dynamische Binäranalyse* und die *Sicherheit von Betriebssystemen* beigetragen.

Abgesehen von den Vorteilen, die virtualisierungsgestützte Sicherheitsdienste mit sich bringen, stehen vorherige Forschungsarbeiten vor einer Reihe von Herausforderungen. Beispielsweise besteht eine allgemeine Einschränkung virtualisierungsgestützter Sicherheitsdienste darin, dass sie nicht spontan *auf Abruf* eingesetzt werden können; Sie erfordern einen kompatiblen Hypervisor, der im Voraus eingerichtet werden muss, um die erforderliche Grundlage zu implementieren. Darüber hinaus verlassen sich moderne Dienste, die auf

der Introspektion virtueller Maschinen basieren, auf hardwaregestützte Funktionen, um technisch anspruchsvolle Malware auf *verdeckte Weise* dynamisch zu analysieren. Obwohl sich diese Fähigkeiten mit der Zeit weiterentwickelt und großes Potenzial auf x86-basierten und insbesondere Intel-Architekturen gezeigt haben, fehlt ihnen die notwendige Grundlage, um auf anderen Architekturen gleichermaßen effektiv zu sein. Dies ist zum Beispiel einer der Gründe, warum ARM bisher trotz seines Durchbruchs und der zunehmenden Akzeptanz auf dem Servermarkt nur unzureichende Aufmerksamkeit im Hinblick auf verdeckte Analyse erhalten hat. Die Motivation, Sicherheitsdienste in isolierte Umgebungen zu verlagern, hat Sicherheitsexperten unbestreitbar zu einer verbesserten Inspektion, Widerstandsfähigkeit und Tarnung in Hinsicht auf dynamische Analyse anspruchsvoller Malware verholfen. Allerdings müssen virtualisierungsgestützte Sicherheitsdienste für Betriebssysteme ihre Logik und Fähigkeiten nicht immer vollständig auslagern. Tatsächlich schränkt diese Strategie die Portabilität virtualisierungsgestützter Sicherheitsmaßnahmen ein, weil sich die ausgelagerten Dienste vom eingesetzten Hypervisor abhängig machen.

In dieser Arbeit untersuchen wir neue virtualisierungsgestützte Techniken, um ihre Kapazität im Hinblick auf die Sicherheit weiter zu verbessern. Um dies zu erreichen gehen wir die oben genannten Herausforderungen an. Insbesondere positionieren wir unsere Forschung um zwei Hauptsäulen, die unsere Arbeit stützen, um den Stand der Technik von virtualisierungsgestützten Grundbausteinen für die *dynamische Binäranalyse* und die *Sicherheit von Betriebssystemen* zu verbessern. Bevor wir uns jedoch den beiden Säulen widmen, beginnen wir unsere Arbeit, indem wir eine Grundlage für beide Forschungsrichtungen aufstellen. Diese Grundlage erlaubt uns beliebige virtualisierungsgestützte Dienste dynamisch—auf Bedarf—bereitzustellen. In diesem Zusammenhang gehen wir die erste der vorgestellten Herausforderungen an, indem wir einen leichtgewichtigen Mikrokernel-basierten Hypervisor, den wir als WhiteRabbit bezeichnen, vorstellen. Wir verwenden WhiteRabbit, um virtualisierungsgestützte Bausteine dynamisch zur Verfügung zu stellen. Wir veranschaulichen ein Szenario, in dem wir einen Dienst zur Introspektion virtueller Maschinen dynamisch unter einem aktiven Linux-System installieren, ohne jegliche Analyseartefakte im Gastsystem zu hinterlassen. Wir zeigen, dass der virtualisierungsbedingte Mehraufwand unseres Prototyps mit führenden Hypervisor Systemen konkurrieren kann und somit eine realistische Alternative zu heutigen Systemen darstellt.

Anschließend wenden wir uns der ersten Säule zu, die sich damit befasst, den Stand der Technik von virtualisierungsgestützten Bausteinen für die dynamische Binäranalyse zu verbessern. Hierfür untersuchen wir die ARM Architektur, um Defizite zu identifizieren und neuartige Techniken zu entwickeln, die für eine effektive virtualisierungsgestützte dynamische Binäranalyse erforderlich sind. Insbesondere entwickeln wir neuartige Bausteine, die es uns ermöglichen auf ARMv7- und ARMv8-Anwendungsprozessoren—speziell auf der AArch32 beziehungsweise AArch64 Architektur—getarnte Software-Breakpoints zu setzen und zu überspringen. Diese Bausteine bleiben für das Gastsystem verborgen und stellen die Basis für eine getarnte dynamische Binäranalyse dar. Um dies zu erreichen, nutzen wir die Virtualisierungserweiterungen des Systems in einer neuen Weise, die es uns ermöglicht die nicht-vorhandene Hardwareunterstützung in Hinblick auf eine

getarnte Binäranalyse zu ersetzen. Hierfür erweitern wir den Xen-Projekt-Hypervisor um die Fähigkeit, dynamisch verschiedene erweiterte (Second Level) Seitentabellen auf ARM zuzuweisen und zwischen ihnen zu wechseln, um verschiedene Sichten auf den Gast-physischen Speicher zu definieren. Nach unserem besten Wissen, sind wir die ersten, die diese Fähigkeit nutzen, um Software-Breakpoints im Speicher des Gastes auf AArch64 zu verstecken. Indem wir zusätzlich den Assoziativspeicher, also den Translation Lookaside Buffer des Systems de-synchronisieren, gelingt es uns, eine getarnte Lösung auf AArch32 zu etablieren. Wir demonstrieren die Effektivität unserer Arbeit, indem wir den dynamischen Binäranalysedienst DRAKVUF mit unseren Bausteinen ausstatten und somit die Grundlage für eine getarnte dynamische Binäranalyse auf ARM ermöglichen.

Für die zweite Säule unserer Forschung nutzen wir Virtualisierungstechniken, um sicherheitskritische Betriebssystemkomponenten zu härten. Wir stellen Bausteine für die Selective Memory Protection (xMP), also für einen selektiven Speicherschutz vor, die Gastsystemen ermöglichen, sensible Daten in isolierten xMP-Domänen im Benutzeradressraum und im Adressraum des Betriebssystemkerns zu isolieren und vor datenorientierten Angriffen zu schützen. In diesem Zusammenhang betrachten wir die Virtualisierungserweiterungen des Systems nicht mehr als Komponenten, die nur dem Hypervisor allein zur Verfügung stehen. Stattdessen binden wir sie als inhärente Bausteine in die Subsysteme des Betriebssystems ein und erlauben ihnen eigene Regeln zu definieren. So ermöglichen wir eine Schnittstelle zwischen der Linux-Speicherverwaltung und Xen, um verschiedene Sichten auf den Gast-physischen Speicher zu definieren. Wir nutzen die unterschiedlichen Sichten auf den Gast-physischen Speicher, um disjunkte xMP-Domänen zu erstellen. Indem wir zusätzlich Zeiger auf Daten innerhalb von xMP-Domänen mit kontextgebundenen HMACs ausstatten, gelingt es uns, datenorientierte Angriffe zu unterbinden. Wir wenden unsere xMP Implementierung auf Datenstrukturen mit Prozessberechtigungen und Seitentabellen im Adressraums des Betriebssystemkerns und auf kryptographisches Material ausgewählter Anwendungen im Benutzeradressraum an. Schließlich zeigen wir, dass die xMP-Bausteine nur einen begrenzten Rechenmehraufwand verursachen und gleichzeitig eine effektive Verteidigung gegen datenorientierte Angriffe darstellen.

Wir schließen unsere Forschung ab, indem wir uns auf die nächsthöhere Abstraktionsebene der Virtualisierungstechnologie konzentrieren. Insbesondere verbessern wir die Sicherheit moderner Virtualisierungstechniken auf Betriebssystemebene (Container) unter Linux. In diesem Kontext stellen wir Jesse, einen auf statischer Analyse basierender Dienst zur Anpassung von Richtlinien für den Linux Secure Computing (seccomp) Modus, vor. Jesse verwendet eine auf abstrakten Interpretation basierende Konstantenpropagierung, um eine Obermenge von Systemaufruf-Nummern zu identifizieren, die die Programme in einem bestimmten Docker-Container aufrufen dürfen. Darüber hinaus kombinieren wir Jesse mit Container- und Bibliotheks-Debloating-Techniken, um nur die notwendigen Bereiche der `libc` Bibliothek einzugrenzen, die Jesse während der Analyse berücksichtigt. Sobald wir die autorisierten Systemaufrufe extrahiert haben, kompilieren wir seccomp-Richtlinien, um den Zugriff auf nicht-benötigte und potenziell verwundbare Systemaufrufe einzuschränken. Durch unseren Prototypen demonstrieren wir, dass Jesse in der Lage

ist existierende Container-Eskalations-Exploits in der realen Welt zu verhindern.

Insgesamt zeigen wir durch unsere Forschung, dass das Potenzial der Virtualisierungstechnologie in Bezug auf die Sicherheit noch nicht vollständig ausgeschöpft ist. Unsere Prototypen zeigen, dass die eingeführten virtualisierungsgestützten Bausteine nicht nur den Stand der Technik in der dynamischen Binäranalyse voranbringen, sondern auch neue Horizonte von Betriebssystem-Sicherheitsarchitekturen eröffnen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Change has never happened this fast before, and it will never be this slow again.
— GRAEME WOOD

Virtualization technology has come a long way since its origin at IBM in 1960s. At that time, IBM implemented CP-40, the first hypervisor that supported virtual memory and full system virtualization for IBM's System/360 mainframes [Bro09]. The CP-40 hypervisor and, in particular, its successor CP-67 was a turning point for IBM, and since then has inspired future technology and paved the way for virtualization, as we know it today. System virtualization has had its peak in the 1960s and 1970s, yet, in the 1980s and 1990s, the lack of computing power has restrained the development of hypervisors for new architectures [Bro09]. In the past two decades, virtualization has experienced its renaissance. The continual technological advances have gained momentum required to increase the computing power to an extent to which virtualization became attractive; virtualization has improved the utilization and energy consumption of the thriving hardware resources, which were otherwise not exploited to their full capacity. Over time, most of the modern application processor architectures added hardware support for virtualization. In fact, today, virtualization is omnipresent and has become an integral element of our everyday lives. Driven by the concept of abstraction, virtualization techniques are embedded in different architectural levels, with each being responsible to implement an isolated and simplified view on the underlying resources [SN05]. One of the most prominent application scenarios is the modern cloud infrastructure that utilizes virtualization technology to improve the availability and overall resource utilization of physical servers; clustering a high number of virtual machines on a small set of powerful physical machines helps distributing and optimally utilizing the available resources that would otherwise not reach their capacities. At the same time, particularly with regard to the increasing complexity of malware and the rising value of data privacy and integrity, the security de-

mands on such infrastructures exploded. Consequently, the rising demands have strongly influenced the research on operating system (OS) security through virtualization, which drives the main focus of this work. To motivate our work, the following summarizes our main research direction, the entailed challenges that we aim to overcome, and our vision of virtualization-assisted OS security that, we believe, will strongly affect the design of future hypervisor and OS architectures.

1.1 Motivation

Exploiting vulnerabilities has never been as lucrative as it is today. The modern world of digitalization and highly interconnected systems has inspired malicious actors and organizations to form a new business model that utilizes a wide range of malware types, with each type designed to target a particular group of interest. Different malware types are classified into different categories based on the applied infection and execution strategy, volatility, and stealth [Vog15, Rut06a]. Often, the concept of sophisticated and stealthy malware hinges on unauthorized behavior modification of high-privileged and security-sensitive software [Rut06a]. The identification of the malicious behavior is often extremely difficult, yet, vital for providing remedies in form of security patches.

To further complicate the situation, malware authors additionally impede static and dynamic analysis by resorting to obfuscation techniques [UPBSB15, JRWM15]. For instance, run-time packers present one of the most commonly applied obfuscation methods for malware. Packers are responsible—contrary to what the term, packer, implies—for unpacking (i.e., reconstructing) the original malware binary at run-time. This process, e.g., decompresses or decrypts the packed code in memory and can involve an arbitrary number of unpacking layers. Each layer is responsible for unpacking another layer until the original binary has been restored. Sophisticated malware packers involve multiple, potentially encrypted layers of unpacking elements that can be interleaved with code sequences of the packed binary, and even distributed among different processes. Further combined with anti-debugging and integrity validation checks [Rut04, CAM+08, BCK+10, SAM14, BBF+16, MANP17], it takes a lot of effort, even for highly skilled reverse engineering experts, to fully comprehend the malicious behavior and intent, and to narrow down the malware's point of entry. Taken into account the time and finances required to detect, analyze, and counteract security breaches renders this an economically unsustainable and almost hopeless situation, without having the right tools at hand.

The growing complexity of modern malware drives security experts to increasingly leverage virtualization technology to assist malware detection and analysis [CN01, GR03, DRSL08, PSE11, GDXJ11, VKSE13, LMP+14, PLM+18]. In fact, almost two decades ago, researchers became aware of the benefits offered by virtualization with respect to security [CN01]. They have suggested to move security applications out of the OS into a high-privileged environment that can be realized through system virtualization. In essence, system virtualization adds a high-privileged software layer, the hypervisor (also known as the virtual machine monitor (VMM)), between the OS and the underlying hardware.[1] The VMM manages system resources and exposes only a narrow virtual hardware interface, the virtual machine (VM), that forms an isolated execution environment for guest OSes. In this way, the VMM bares only a limited attack vector, yet, at the same time, it maintains a

[1]Throughout this work, we use the terms *hypervisor* and *VMM* interchangeably to describe the high-privileged virtualizing software layer that implements a virtual hardware interface, beneath which it transparently manages hardware resources on behalf of guest OSes [SN05].

complete and untainted view over the entire VM's binary and register state [PG74, Pfo13].[2]

These properties have inspired researchers to form novel virtualization-assisted techniques for security, which they have coined virtual machine introspection (VMI) [GR03]. VMI assists security experts to analyze and manipulate the state of guest OSes from the outside. The VMM's strong isolation capabilities ensure that even compromised VMs cannot manipulate the virtualization-assisted monitors, which can operate from a different VM or from the VMM itself. More importantly, virtualization lends external monitors an omniscient character and inherently hinders malware inside VMs from deluding the analysis, and thus has become indispensable for malware analysts and reverse engineers.

Over time, VMI techniques have proven effective. In particular the rich variety of hardware-supported virtualization extensions of the x86 architecture—with a strong emphasis on Intel—has given research on VMI the necessary ground for inspiration and progress. Yet, to benefit from these techniques, a VMI-aware VMM must be set up in advance underneath the target system; a constraint for the massive application of VMI. In other words, systems that were not explicitly set up for VMI cannot benefit from these capabilities. Without VMI, to perform memory forensics, reverse engineers would need to resort to complex data-recovery methods to extract memory contents by relying on potentially compromised OS mechanisms. To break through the given constraint, one must call into question the rigid deployment strategy of VMMs. Specifically, if we recall sophisticated virtualization-based rootkits that employ on-the-fly virtualization techniques to subvert OSes [Rut06b, RT07, Zov06], the question arises whether we can adopt similar techniques to deploy thin VMMs to facilitate VMI and other virtualization-assisted frameworks on-demand. Besides, inspired by the ingenuity of attackers, one should investigate the benefits of such on-the-fly deployment strategies and question whether or not other hardware architectures can benefit from similar mechanisms.

Regardless of the VMM deployment strategy, virtualization has helped defenders to gain the upper hand in the never-ending arms race against the malicious actors. In fact, over the past few years, there has been a substantial rise in adoption of virtualization technology in analysis frameworks in both commercial [VMR20, Fir20, Bit20] and open-source applications [PSE11, VKSE13, LMP+14, Lib20, Vol20]. Facing the new challenge, attackers have quickly adapted and equipped malware with anti-virtualization techniques [CAM+08, BCK+10, SAM14, MANP17], which scrutinize the execution environment for artifacts to reveal and evade VMI-based analysis frameworks. This feature has lent malware a split personality that actively avoids revealing its logic to security experts and hence poses a new challenge for reverse engineers. This situation, in turn, has formed a new incentive for VMI researchers to investigate a novel property, namely *stealth*, that has not received sufficient attention in the past. Even though perfect VM transparency (i.e., the ability of being undistinguishable with real hardware) is infea-

[2] We disregard AMD Secure Encrypted Virtualization (SEV), AMD SEV Encrypted State (SEV-ES), and AMD SEV Secure Nested Paging (SEV-SNP) extensions to encrypt the guest's memory (and register state) to make them inaccessible to a benign, yet, potentially vulnerable VMM.

sible in practice [GAWF07], the modern trend towards system consolidation through virtualization renders the goal of VM transparency obsolete. Discrepancies between physical and virtual machines cannot completely hide the presence of a virtual environment [Rut04, CAM+08, Pfo13, SAM14, MANP17, TBZ+18].

At the same time, server virtualization is becoming omnipresent. Thus, a virtualized system does not necessarily indicate that its sole purpose is malware analysis. Therefore, it does not make economic sense for attackers to exclude virtual environments. Nevertheless, malware can still detect VM-based analysis systems, through artifacts, ranging from guest-accessible memory to register contents. Contrary to VM transparency, this highlights the value of the stealth property. Previous research has shown that VMI-based analysis frameworks can achieve a high degree of stealth [PSE11, Pfo13, LMP+14] by means of modern hardware virtualization extensions of the x86 architecture. Contrary to x86, other architectures struggle to foresee hardware capabilities that facilitate, in particular, stealthy VMI. This places a great emphasis on the question of how to conceal VMI-based analysis frameworks on architectures without the necessary hardware support.

One concrete example is given by the ARM architecture. ARM has become the leading processor architecture for mobile, wearable, and Internet of Things (IoT) devices. Through the continuously increasing computing performance and the added hardware support for virtualization, ARM has recently started claiming a bigger slice of the server market pie as well [Ama20, The20b]. As such, it will not be long before malware starts more regularly targeting the ARM architecture. Therefore, the stealthy operation of VMI on ARM is an obligation to successfully analyze and proactively mitigate this growing threat. Stealthy VMI has proven itself perfectly suitable for malware analysis on x86—the dominant player in the virtualization and server industry—, yet, it lacks the foundation required to be equally effective on ARM [PLM+18]. While previous work employs VMI techniques on ARM for security purposes [YY12, GVJ14, TKFC15], it does not emphasize its stealthy nature. Consequently, cloaking analysis frameworks remains an open question and emphasizes the need for novel primitives, which empower stealthy monitoring on ARM.

Until recently, the virtualization extensions of the system's Instruction Set Architecture (ISA) have served the sole purpose of supporting the VMM in managing, isolating, and efficiently distributing the system's hardware resources among different VMs. With VMI, this perspective has changed; virtualization extensions are used to facilitate powerful malware detection and analysis systems. Yet, we claim that the capacity and possibilities provided by virtualization have not yet been explored to their full extent. From this vantage point, we believe that modern OS and VMM architectures require a paradigm shift, in which services of the VMM can become an integral part of the OS. Our vision is to integrate, in particular, the strong memory isolation capabilities of the virtualization extensions into the guest to assist critical elements of the OS. Consequently, in this work, we investigate entirely new ways of repurposing virtualization extensions to define in-guest policies, which are enforced by the underlying VMM. This work further considers ways of supporting the security of OS-level virtualization and ultimately sets the ground for novel in-guest primitives that can form a strong defence in kernel and user space.

1.2 Research Questions

This work intends to investigate and highlight the capabilities of virtualization techniques that have not yet been explored to their full potential with regard to security. Specifically, we approach this high-level goal from two different security perspectives, in which we leverage virtualization for (i) *stealthy dynamic binary analysis* and (ii) *OS security*. To accommodate both perspectives, we form the following two main research questions, the answer to which renders the main focus of this work. The first question addresses how to employ virtualization to improve the state-of-the-art foundation for dynamic binary and malware analysis techniques. The second question considers aspects of virtualization that can assist modern OSes in guarding sensitive components against sophisticated attacks. To complement both questions, we take into account the x86 as well as the ARM architecture.[3] Overall, our key research questions can be further subdivided into the following concrete objectives:

(Q1) How can we realize stealthy VMI on the ARM architecture? With ARM becoming one of the most prevalent architectures for mobile, wearable, IoT, and now even in the server market, it presents an attractive target for malicious actors. We have to investigate whether the architectural capabilities of modern ARM virtualization extensions meet the demands of VMI-based malware analysis tools that have proven effective on x86. In particular, the question arises to what extent the ARM architecture can ensure stealthy VMI and what means are necessary to empower it?

(Q2) How can we apply virtualization to enhance the security of OS components? Given the strong memory isolation capabilities of virtualization, the first question that comes into mind is which OS components can benefit from this competence. To answer this question, we must determine and investigate potential hardware-architectural deficits that struggle protecting against certain attack patterns. Assuming there is a need for novel in-guest memory isolation primitives that leverage hardware's virtualization extensions, we have to answer the question which architectural modifications of the modern OSes and VMMs are necessary to empower the OS subsystems with the new capabilities to promote virtualization-assisted security on different software levels.

(Q3) How can we assist OS-level virtualization to enhance its isolation capabilities? The global accessibility and flexibility of modern OS-level virtualization solutions (i.e., containers) has gained popularity. Containers share the OS kernel with the host, yet, unlike system virtualization techniques, they lack strong isolation capabilities. Still, the communitity utilizes containers not only for convenient packaging of services but also for security-critical tasks. At this point, the question one has to answer is whether and how we can enhance the isolation of containers to improve their overall security.

[3] Throughout this work, we refer to both x86 and x86-64 as the x86 architecture.

1.3 Thesis Statement and Contribution

Thesis statement: virtualization-assisted security has not yet been explored to its full extent; modern hardware-supported virtualization extensions supply the necessary foundation to (i) establish stealthy primitives for dynamic malware analysis frameworks on ARM architectures (despite their hardware deficits for stealthy monitoring), and (ii) transform the design of modern OS architectures such that they can leverage the strong memory isolation capabilities of the virtualization extensions to enhance the security of its subsystems.

The following summarizes the concrete contributions of this work, which address the identified key research questions and provide a glimpse into the true potential of virtualization technology with regard to security.

We propose a flexible VMM deployment strategy to facilitate on-demand deployment of VMI frameworks and virtualization-assisted security primitives for OS subsystems. We begin our work by investigating on-the-fly virtualization techniques to overcome the rigid nature of VMM deployment. Specifically, we lay out a minimalistic microkernel-based VMM architecture that leverages on-the-fly virtualization techniques for deployment on x86 and ARM architectures. The main objective of the new VMM architecture is to dynamically and transparently shift a running OS into a virtual environment, an inspiration that we borrow from the Blue Pill rootkit for x86 [Rut06b, RT07]. Yet, as so often in security, we transform the originally malicious intention behind this technology for the benign purpose of equipping virtualization-assisted security-driven frameworks with a flexible deployment strategy; this strategy facilitates periodic, sporadic, as well as permanent—yet retrospective—deployment of only the absolutely necessary virtualization components, avoiding immense, error-prone, and, for many purposes, irrelevant code base of traditional VMMs. Once deployed, our architecture allows to transparently unfold an extensible, microkernel-based architecture underneath the dynamically virtualized OS. In this way, we broaden the capability horizon of the OS, in particular, through strong and fine-grained memory isolation features that are otherwise not available to the OS.

We support our design with a prototype, WhiteRabbit, that dynamically virtualizes a running Linux by leveraging hardware virtualization extensions of the x86 architecture. We highlight that even though our prototype focuses on dynamically and transparently deploying a VMI infrastructure for local and remote VMI frameworks, it can similarly deploy the necessary means required to assist the deployment of functionality that assists in-guest security-relevant subsystems of the dynamically virtualized OS.

Impact: By exploring this research direction, we establish a flexible, on-demand deployment strategy that can act as a vehicle for VMI frameworks as well as virtualization-assisted in-guest security subsystems on x86 and ARM. In fact, we can apply this strategy to many techniques that we propose throughout this work. Further use cases include protecting, monitoring, and forensic analysis of managed corporate and IoT infrastructures.

Publication: Parts of this contribution have been published in the paper [PKZ18].

We identify and overcome deficits of the ARMv7 and ARMv8 architectures to facilitate novel primitives for stealthy virtualization-assisted malware analysis on ARM. The hardware-supported virtualization extensions on Intel form a solid foundation for stealthy VMI. Inspired by Intel, we define a set of requirements that are necessary to establish stealthy monitoring on the ARM architecture. In this regard, we have identified that, contrary to Intel, the ARMv7 and ARMv8 architectures lack sufficient hardware capabilities required for stealthy VMI. (While ARM generally does not support *stealthy single-stepping*, its 32-bit execution state, AArch32, in addition, has no notion of *execute-only* memory pages; every code page has to be *readable* and *executable*.) As such, we shift our focus towards exploring and developing new primitives that empower ARM for stealthy malware analysis, despite the spotted architectural shortcomings. To compensate unmet requirements for stealthy monitoring, as a first step, we propose a novel single-stepping primitive, without using the hardware-intended functionality—which by itself can reveal the analysis framework. Further, we equip the Xen project hypervisor with the ability to dynamically allocate and switch among different guest-physical memory *views*. Specifically, we implement the Xen *alternate p2m* (`altp2m`) subsystem for the ARM architecture. This subsystem allows an external monitor to switch among a set of second level address translation (SLAT) tables to enforce different memory access permissions of individual guest-physical page frames per view and remap individual guest frames to different machine frames. That is, instead of switching permissions of individual guest frames in one global view, Xen `altp2m` allows us to switch among different views—immediately changing the guest's perspective on the guest-physical memory.

Impact: By combining both of the above methodologies, we form the basis for stealthy VMI on AArch64. In addition to that, we tackle the architectural deficit of AArch32 with respect to *execute-only* memory; we utilize the Xen `altp2m` implementation in a way that allows us to take control of the Translation Lookaside Buffer (TLB) organization on AArch32. By de-synchronizing the TLB organization, we manage to hide instrumented code pages in the guest's memory and realize the foundation for stealthy VMI on AArch32. Finally, we develop and open source the foundation for the binary analysis framework DRAKVUF [LMP+14] and hence establish the means for stealthy dynamic malware analysis on both AArch32 and AArch64.

Publication: Parts of this contribution have been published in the paper [PLM+18].

We integrate virtualization capabilities into guest OSes to establish in-guest memory isolation primitives for protecting sensitive data against data-oriented attacks. Having built a solid foundation for stealthy VMI on ARM architectures, we shift the focus of our research on virtualization towards empowering guest OSes with additional capabilities with respect to security. In this regard, we investigate and leverage the unique properties of the modern Intel virtualization extensions to promote Extended Page Table (EPT) management tasks to the guest OS. Combined with the Xen `altp2m` subsystem, we utilize the Intel *EPT pointer (EPTP) switching* capability to manage different views on the

guest-physical memory from inside a VM. In this way, we lend the guest the capability to interact with the system's virtualization extensions, without any hypervisor interaction. We extend the memory management system of the Linux kernel to realize *xMP*, an in-guest virtualization-assisted *selected memory protection primitive* for guarding sensitive data against data-oriented attacks in both user and kernel space. Specifically, xMP combines Intel's EPTP switching and the Xen `altp2m` subsystem to control different guest-physical memory views, to isolate sensitive data in disjoint xMP domains. We equip in-kernel management information and pointers to sensitive data in xMP domains with authentication codes, whose integrity is bound to an immutable context to impede data-oriented attacks. In a more concrete application scenario, we apply xMP to guard all page tables and process credentials in the Linux kernel, as well as sensitive data in user-space applications, with only minimal performance overhead. Finally, we integrate xMP into the Linux namespaces framework, and hence form the foundation for virtualization-assisted OS-level virtualization (container) protection against data-oriented attacks.

Impact: This contribution establishes the foundation for *virtualization-assisted security* mechanisms that allow to alleviate the strict separation between an OS and a VMM. The implemented use cases demonstrate the flexibility and power of the introduced concepts by hardening different subsystems of the Linux kernel, user space processes, as well as containers against data-oriented attacks. The virtualization-assisted security primitives are building blocks, which extend the capabilities of the OS to leverage the system's virtualization extensions for security, without having to outsource security-relevant logic into the VMM. In other words, contrary to VMI techniques, any VMM that implements a common Application Programming Interface (API) for virtualization-assisted security can assist in-guest security subsystems in a way that is completely indifferent to the semantic knowledge of the OS.

Publication: Parts of this contribution have been published in the paper [PMG+20].

We support OS-level virtualization by statically generating last line of defense seccomp policies to limit the number of system calls available to Docker containers. In addition to isolating selected OS resources through xMP namespaces from potentially compromised containers, we introduce Jesse, a static analysis based framework for generating Linux seccomp policies for non-obfuscated binaries in Docker containers. The system call interface of modern OSes offers user space applications access to a high number of system calls. Unfortunately, a vulnerability in one of those system calls can pave the way for attackers to compromise the entire OS kernel. By following the *principle of least privilege*, with Jesse, we aim at granting containers access only to those system calls that are absolutely necessary for their genuine execution. Jesse statically generates policies for the seccomp facility on Linux that has the capability to filter out unused and potentially vulnerable system calls. Specifically, Jesse implements an abstract interpretation based constant propagation for conservatively connecting system calls to identified system call invocations (`syscall` instructions) in binaries. By additionally applying dead code elimination techniques to

used general-purpose libraries, such as `libc`, we make sure that the generated system call filters consider only those system calls that are relevant for the container's execution.

Impact: Even though the proposed implementation targets ELF binaries in Docker containers compiled for the x86 architecture, its general concepts are independent of the architecture and are similarly applicable to all binaries of the system.

Publication: At the time of submission, parts of this contribution have been accepted and are to appear in the paper [GPZ23].

1.4 Publications

Parts of the contributions in this work base upon the following published, scientific, and peer-reviewed articles. Not all parts of our work were published before the date of submission of this thesis. Work that has been submitted for review is stated as such at appropriate locations.

[PKZ18] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection. In *IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, 2018.

[PLM+18] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Inrospection. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[PMG+20] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[GPZ23] Charlie Groh, Sergej Proskurin, Apostolis Zarras. Free Willy: Prune System Calls to Enhance Software Security. To appear in *ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2023.

The following published, scientific, and peer-reviewed articles comprise further contributions that have not been considered in this work.

[PMK12] Sergej Proskurin, David McMeekin, and Achim Karduck. Smart Camp: Building Scalable and Highly Available IT-Infrastructures. In *IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, 2012.

[PKE15] Sergej Proskurin, Fatih Kilic, and Claudia Eckert. Retrospective Protection utilizing Binary Rewriting. In *Deutscher IT-Sicherheitskongress*, 2015.

[PWS15] Sergej Proskurin, Michael Weiss, and Georg Sigl. seTPM: Towards Flexible Trusted Computing on Mobile Devices based on GlobalPlatform Secure Elements. In *International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 2015.

[MPR+21] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. SEVerity: Code Injection Attacks Against Encrypted Virtual Machines. In *IEEE Workshop on Offensive Technologies (WOOT)*, 2021.

At the time of submission of this dissertation, the following work remains under review:

Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Vasileios P. Kemerlis, and Michalis Polychronakis. Immutable Memory Management Metadata for Commodity Operating System Kernels. *Under review*, 2022.

1.5 Outline

The remainder of this work is organized as follows. In Chap. 2, we summarize the foundations regarding virtualization technology to assist the reader in better understanding the course and details of our work. In this chapter, we outline relevant components of the hardware virtualization extensions, and conclude with a discussion about the benefits and challenges of VMI. We then proceed with Chap. 3, in which we put the perspective of our research directions and the individual objectives into a concrete frame. In this regard, we introduce a conceptual, hardware-independent target system architecture that allows us establish a bird's eye perspective upon the overall picture of the contributions of this work. We further present a comprehensive, and generic threat model describing the adversarial capabilities targeting the abstract system. We extend our assumptions of the generic threat model for each framework presented in the following chapters.

Having laid out the necessary foundation, we proceed with Chap. 4 – 7, which present the main pillars of our research. Yet, before we begin addressing our research objectives (Sec. 1.2), in Chap. 4, we elaborate an alternate, on-demand deployment strategy for virtualization-assisted services. Specifically, we present WhiteRabbit, a thin microkernel-based VMM that can be deployed on-the-fly. We exemplify a specific use case, in which we apply WhiteRabbit to deploy a VMI-based framework on a running Linux. At the same time, we underline WhiteRabbit's ability to act as a generic vehicle, which is not limited to VMI, and hence can be leveraged to dynamically deploy any of the virtualization-assisted services of the following chapters. We conclude this chapter by assessing and comparing the virtualization overhead of WhiteRabbit with Linux KVM and the Xen hypervisor.

Further, in Chap. 5, we focus on improving the state-of-the-art VMI techniques on ARM. In this chapter, we identify that ARM does not provide the necessary hardware support in regard to stealthy monitoring. To compensate the missing hardware capabilities, we explore alternative directions for VMI. We repurpose the system's virtualization extensions to implement novel primitives for setting and single-stepping software breakpoints in a stealthy way, without using the intended hardware mechanisms. We integrate the presented strategy into the dynamic binary analysis framework, DRAKVUF, and hence empower ARM with the ability to stealthy monitor guest OSes. We conclude this section by extensively assessing the performance and effectiveness of the introduced primitives.

In Chap. 6, we turn our attention towards reinforcing the security of OS components through virtualization. We leverage virtualization extensions on Intel to establish selective memory protection (xMP) primitives, which we intend to use against data-oriented attacks in kernel and user space. We equip the Linux memory management system with the ability to configure and switch among different views on the guest's physical memory. By configuring the views in a sophisticated way, we establish the means for protecting sensitive data in kernel and user space in isolated and disjoint xMP domains. In addition, we equip in-kernel management information and pointers to sensitive data in xMP domains with context-bound authentication codes to obstruct data-oriented attacks. We conclude this section by assessing the performance and security of the introduced xMP primitives.

In Chap. 7, we focus on enhancing the security of OS-level virtualization (i.e., containers) techniques. Contrary to the previous chapters, we utilize the Secure Computing (seccomp) mode of the Linux kernel to reduce the system call interface. In this way, we intend to eliminate unnecessary and potentially vulnerable system calls, which would be otherwise freely available to the containers. Specifically, we introduce Jesse, a framework that combines a set of static analysis techniques to tailor seccomp policies for (containerized) Executable and Linkable Format (ELF) binaries. Jesse either extracts binaries from container images (or directly takes them from the file system) and applies an abstract interpretation based constant propagation technique to identify authorized system calls for a given binary. Additionally, we combine Jesse with state-of-the-art library debloating techniques to associate each of the exported functions in the standard C library (`libc`) with a set of system calls it requires for its execution. In the end, Jesse combines the identified system calls of the analyzed binary with the system calls of each `libc` function the analyzed binary requires. Finally, Jesse uses the extracted system calls to generate seccomp policies for the respective Docker containers. We conclude this chapter by evaluating the precision of our analysis and by applying Jesse to popular Docker images and protecting them against real-world container escalation exploits.

We conclude this dissertation in Chap. 8, in which we summarize our contributions and provide an outlook on future research directions with regard to enhancing the state of virtualization-assisted analysis and OS security.

Chapter 2

Technical Background

If you can't explain it simply, you don't understand it well enough.
— Albert Einstein

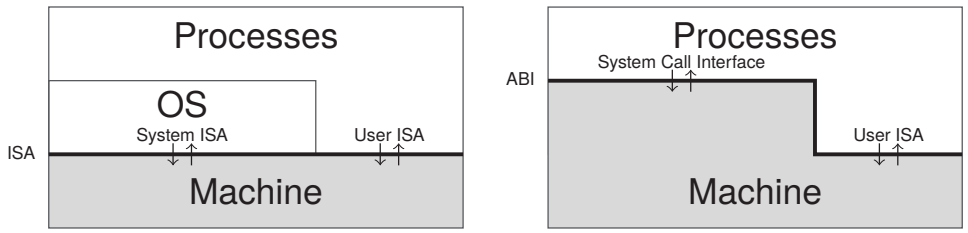
This chapter assists the reader with the fundamentals that are essential to the understanding of the remainder of the dissertation. In particular, the following sections cover different concepts of the virtualization technology, the necessary means for their implementation, and the benefits and challenges behind VMI. In other words, the conveyed knowledge of this chapter renders the necessary foundation, upon which we base our research on virtualization-assisted dynamic binary analysis and OS security.

2.1 Virtualization Technology

The adoption of virtualization techniques in modern computer systems has become omnipresent and unavoidable. In fact, virtualization is deeply embedded in various computer system components, which we often do not consciously, or directly associate with virtualization itself. For instance, even though *virtual memory* is regarded as a given construct, it represents one form of virtualization that allows to decouple, or abstract the management of the limited physically available memory resources of one or multiple interconnected systems. This form of virtualization contributes to better performance and overall physical memory utilization, as well as security through address space isolation. Further examples for virtualization comprise among others: *virtual system calls* [Bov14a] which increase the performance of selected system calls by emulating their functionality in user space; and the *extended Berkeley Packet Filter (eBPF) in-kernel virtual machine* which establishes an execution environment for user-supplied programs in the Linux kernel, e.g., for filtering network traffic or unauthorized system calls, tracing, and in-kernel optimization [Fle17]. While these virtualization techniques transparently assist us in our every-day life, in this thesis, we mainly focus on virtualization techniques that establish isolated execution environments for individual (or groups of) processes or entire OSes.

One of the key concepts of virtualization is its ability to manage complexity [SN05]. To fully understand even the most complex systems, we can break down their complex structure into a hierarchy of different levels of *abstraction*; higher, more abstract levels build upon, yet, do not involve with the implementation details that are encapsulated by lower abstraction levels. Instead, the abstraction levels are separated by *well-defined interfaces* that hide complex implementation details. At the same time, the interfaces impose a strict specification; software written for one interface will not work on another. As a result, components residing at higher levels of abstraction receive a simplified view over the lower components in the hierarchy, yet, have to adhere to the specific interface.

To clarify the general idea, let us apply the concept of managing complexity to the above scenario of virtual memory. Virtual memory hides the complexity of managing the physical memory behind a well-defined memory management interface. That is, the virtual address spaces presented to processes—that we regard as system components at a higher abstraction level—embody an abstract, or simplified view of the underlying memory management subsystem at the heart of the OS kernel. The well-defined abstraction interface, relaxes the constraints of the underlying implementation details. In fact, it provides an illusion to user space processes of possessing exclusive access to an immense amount of memory resources, defined by the virtual address space. Specifically, this interface hides the underlying memory management system implementation that is responsible, amongst others, for governing physical memory, page tables, memory overcommitment, and swapping. At the same time, the interface strictly defines how the processes can interact with the virtual address space. By following the specifications of the interface, processes can execute, access, and even allocate and release additional memory, without any concern about these and further details of the overall architecture.



(a) The ISA interface defines the machine. (b) The ABI interface defines the machine.

Figure 2.1: Different interfaces define different perspectives on the machine [SN05].

Similarly, the inherent properties of physical computer architectures define strict and immutable interfaces. Two interfaces that we consider relevant for this work are the *Instruction Set Architecture (ISA)* and the *Application Binary Interface (ABI)* (Fig. 2.1).

Instruction Set Architecture: The ISA represents the lowest interface that interlinks the hardware with the software. It defines the architecture's instruction set, registers, (virtual) memory, and the interrupt and exception architecture. Fig. 2.1(a) illustrates a simplified representation of this interface. In the figure, we further distinguish between components of the *user ISA* and *system ISA*, which are visible to less- or high-privileged software, respectively. While the user ISA is responsible for establishing an execution environment for software, the system ISA assists the OS in managing the system's hardware resources. The microarchitecture of different Central Processing Units (CPUs) ensures that software, developed for a specific ISA, is compatible to all CPU variants that implement the same ISA. Thus, the ISA is responsible for software compatibility and strictly regulates which software components are authorized to access specific parts of the underlying system.

Application Binary Interface: Similar to the ISA, we can subdivide the ABI into the *system call interface* and the less-privileged *user-visible part of the ISA*. Fig. 2.1(b) sketches this division. Through the system call interface, OSes offer less-privileged user space applications a uniform link that governs access to the OS internals. In other words, the system call interface lends processes in user space the ability to request and communicate with system resources, such as memory and I/O devices.

To sum up, the ISA strictly defines an interface between the OS and the underlying hardware, binding the OS to the particular architecture (Fig. 2.1(a)). Similarly, the ABI separates user space applications from the details implemented by the OS and hardware, ultimately creating dependencies between the applications and the overall platform (Fig. 2.1(b)). In both cases, the respective interface hides the complexity of the implementation details underneath the particular interface, lending software (at higher abstraction levels) an abstract view of the underlying *machine* upon which it is executing. In this regard, an OS perceives the machine through the angle that is defined by the ISA; whereas from the perspective of

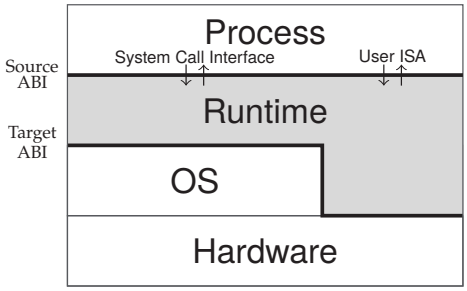


Figure 2.2: Process VM runtimes implement and expose a virtual ABI to processes [SN05].

user space applications, it is the ABI that sheds light on the platform—which comprises the OS and user ISA—that is considered a machine. Consequently, a software interface, which complies to the expected behavior of a particular machine, has the power to provide an illusion of the imitated machine to software that executes on higher abstraction levels. This presents the main idea behind virtualization technology.

Generally, virtualization decouples from the physical hardware constraints by implementing a flexible, virtual interface that is not bound to the hardware. This interface, implements a dedicated virtualization layer that mimics the characteristics of a real machine to form a virtual execution environment, the virtual machine (VM). We refer to the software that executes in the VM as the *guest*, and the system hosting the VM as the *host*. Virtual resources that are presented through the virtual interface to the *guest* are either completely emulated in software or directly mapped to physically available resources of the *host* system [PG74]. Depending on the virtual interface implementation, we distinguish between *process-level VMs* and *system-level VMs*, which, as the names suggest, define execution environments for processes or OSes, respectively. In addition, recent technological advances have introduced *OS-level virtualization*, also known as containers, which leverages OS's services to form a virtual environment for groups of processes. In the following sections, we introduce these concepts and relate them to security and our work.

2.1.1 Process-level Virtualization

Process-level, or simply process VMs form virtual execution environments for a single, or in some cases, a group of processes [SN05]. Fig. 2.2 illustrates a simplified process VM architecture that adds a software virtualization layer that borders the host's ABI. When referring to process VMs, this virtualization layer is called *runtime*. The runtime implements the necessary means to establish a virtual ABI that is compatible to the virtualized process. That is, the complex implementation of this compatibility layer hides beneath the virtual (source) ABI. To establish a consistent taxonomy, we pick up on the introduced notion of complexity management: from the perspective of a process, the abstract view of the *machine* is presented through the ABI, which comprises the user-visible part of the ISA and the OS system call interface (Fig. 2.1(b)). Generelly, since the combination of a

particular ISA and OS is referred to as *platform*, throughout this work we use the term platform to describe the machine as it is perceived by processes.

According to this definition, the host's platform itself can be considered one of the most rudimentary process VMs that implements the concept of multiprogramming (and multitasking); a concept that lends processes the ability to share resources and execute (virtually) at the same time, without being aware of each other. An executing process builds upon the host's ABI and has the impression of having the entire *machine* for its own purposes. Under the hood, the OS manages the resources on behalf of the process, isolates its address space, and time-shares the hardware among different, similarly unaware processes to provide virtual and isolated execution environments. Other key benefits of using process VMs mainly comprise compatibility, but also performance optimization and platform independence.

Compatibility: To establish compatibility, process VMs can expose a *source ABI* that does not necessarily have to comply with the *target ABI* of the underlying platform. That is, process VMs can allow programs compiled for a specific platform to run on a system hosting a different OS or implementing a different ISA. In both cases, the runtime must apply emulation techniques (e.g., through interpretation or binary translation, the concepts of which are not the focus of this work [SN05]) either to translate source instructions into target instructions or to implement the operation of the guest's system calls through the host-supplied functionality.

Performance optimization: To increase the overall performance, dynamic binary optimization frameworks identify hotspots by dynamically gathering run-time information. Based on the gathered execution profiles, such frameworks translate the selected source instructions into optimized target instructions of the same (or different) ISA by means of software or hardware emulators.[1] The binary optimizers cache the translated code to increase its performance upon the next execution.

Platform independence: Finally, a special application scenario for process VMs focuses on establishing platform independence. So called High-level Language (HLL) VMs, with Java Virtual Machine (JVM) and Dalvik Virtual Machine (DVM) being two of the most prominent implementations, define runtimes that establish a platform-independent, virtual ISA. The virtual ISA aims to reduce dependencies to the host's OS and ISA. Consequently, applications compiled against this virtual ISA can execute on any system, as long as the runtime implementation supports the host's platform.

[1] Modern x86 CPU microarchitectures translate CISC into RISC-like μ-operations to increase performance. Further examples are given by ARM Jazelle [Por05] and NVIDIA Denver architecture [BBTV15] that use hardware extensions to translate Java bytecode and ARM instructions, respectively.

Two of the most prominent and noteworthy examples for process VMs, which implement a compatibility layer for processes that were compiled for a different OS, are presented by `Wine` [Win20] and Microsoft Windows Subsystem for Linux (WSL) [Mic16b, YIRS17]. `Wine` implements a runtime that facilitates executing Portable Executable (PE) binaries compiled for Windows on POSIX-compliant OSes, including Linux and macOS. On the other hand, WSL allows to execute Executable and Linkable Format (ELF) binaries on Windows.

On a technical level, `Wine` comprises a runtime that replaces the (user space) system programming interface, namely the Windows API, with a custom implementation that mimics the intended functionality on Linux.[2] Windows leverages a different concept that shifts the runtime into the kernel space. Specifically, Windows applies *pico processes*, which wrap and execute the ELF binaries in user space, and *pico providers*, which are the complementary components that control the state and execution of pico processes from the kernel space [Mic11, Mic16a, YIRS17]. That is, pico providers intercept system calls of pico processes and emulate, or mimic their behavior through the given functionality of the host to provide the illusion of the expected platform.

An example for a process VM framework that allows to execute binaries compiled against a different ISA is given by the Linux kernel support for miscellaneous binary formats, `binfmt_misc` [Cor16]. This functionality allows to register almost arbitrary binary formats (potentially compiled against a different ISA) and the associated interpreters by means of the virtual filesystem `/proc/sys/fs/binfmt_misc/`. Upon executing a binary with a previously registered format, the system will match the binary's magic byte sequence and use the associated interpreter that is capable of emulating the binary's ISA, regardless of the host's platform. In this way, it becomes possible to execute binaries compiled, e.g., for ARM on the x86 architecture (yet by using the same OS).

With regard to security, process VMs provide only light-weight isolation capabilities (e.g., through virtual memory that prevents corrupted processes from directly accessing the memory of other, benign execution environments). Thus, to meet the modern demands on security, process VM runtime implementations apply additional *access control* and *logical separation* techniques to isolate access to global resources—and thus to restrain the attackers [App20]. While most of these techniques are not directly related to our work, there exist a special class of process VMs that raises our attention and affects our research. This class of process VMs, coined *OS-level virtualization*, leverages the OS kernel's features to establish, in particular, *isolated* execution environments for groups of processes. Specifically, OS-level virtualization techniques maintain processes in so called *containers* with a limited view on global system resources. We dedicate the following section to describe modern OS-level virtualization techniques to establish the foundation that intends to assist the reader in understanding some of our design choices.

[2] Even though `Wine`—being the acronym for `Wine` Is Not an Emulator—resists in being associated with emulators, it uses a form of emulation that mimics the semantics of the original Windows API.

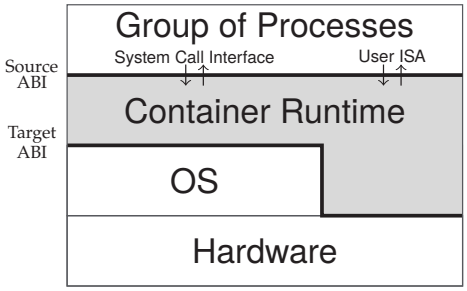


Figure 2.3: Container runtimes lend processes an isolated view on the OS's resources.

2.1.2 OS-level Virtualization

Modern OS-level virtualization techniques leverage services, or rather subsystems, of the underlying OS kernel to establish light-weight isolated execution environments, *containers*, for processes. The first container implementations [Ltd20, SPF+07, Rio20, Ora20] have their origin in Unix-based OSes, and today, they have regained popularity on Linux [Doc20a, Inc20, Fou20, Goo20a] and started to become increasingly attractive on Windows [Fer19]. Given the container's affinity to Linux, the following description focuses on the most fundamental concepts of the Linux kernel that define the basis for OS-level virtualization. In addition, we shed light on selected concepts that facilitate containers on Windows.

A container comprises one or a set of regular user space processes and utilizes the OS kernel to restrict the processes' capabilities, and their perception and access to global system resources. Containers execute directly on the OS kernel, which is responsible for enforcing these restrictions. In contrast to process VMs, compatibility is not the main purpose of the runtime of OS-level VMs (the *container runtime)*. In other words, the container runtime itself does not intend to intercept or emulate the virtualized application's instructions or system calls to establish, e.g., cross-OS compatibility. Instead, the main purpose of the container runtime aims to create efficient, scalable, reproducible, and isolated execution environments with governed resources (Fig. 2.3). Note that other layers of the OS kernel can still pursue the compatibility property. For instance, similar to the WSL process VM [Mic16b, YIRS17] that allows to execute ELF binaries on Windows (Sec. 2.1.1), dedicated abstraction layers of the Windows kernel can complement the container runtime to allow executing Linux containers on Windows.

The container runtime configures subsystems of the underlying OS kernel to establish a contained execution environment that receives its own view on the OS's resources. This is accompanied by a container image that provides a container with its own plumbing layer (i.e., the user space components of the OS). That is, even though different Linux containers share the host's OS kernel, they assemble memory images with their own user space components (including the service to be contained, additional tools, libraries, and a file system hierarchy). In this way, container images present stand-alone software packages that convey the look and feel of a particular Linux distribution. As such, a container's Linux distribution does not necessarily have to be the same as the one used by the host

and can be independently deployed to other systems.

Even though different container runtime implementations vary with regard to high-level functionality (e.g., container image management and deployment), at the lowest level, their core tasks eventually boil down to configuring the OS kernel's sandboxing features.[3] These features can be distributed across three classes that describe the key properties of OS-level virtualization: *resource isolation*, *resource governance*, and *access control*. In the following, we outline the key concepts of each class and support them with an a specific Linux subsystem. Note that although the highlighted concepts are specific to Linux, their general functionality assembles essential building blocks, and is indispensable for containers.

Resource isolation: In the scope of OS-level virtualization, resource isolation lends processes in containers an abstract perspective on selected global system resources. In other words, this property establishes an environment, which creates an illusion of having exclusive access and control of the particular system resources. The Linux kernel implements a set of *namespaces* [Ker13b] to provide such contained environments (i.e., containers) which, when combined, affect the containers' perception such that they believe they are being executed upon a private OS instance; processes in different containers use services of the same OS kernel, yet, they are isolated and cannot observe or (directly) interact with each other.

Resource governance: Linux leverages the *control groups (cgroups)* [Bro14] subsystem to efficiently govern the system's hardware resources (such as CPU, memory, and devices) between containers. Specifically, the Linux cgroups subsystem is a concept that organizes processes (e.g., inside a container) in a hierarchy to form a control group, accounts for, limits, and imposes control upon system resources of this group. To govern resources, the cgroups subsystem applies so called resource controllers to cgroup hierarchies; processes assigned to a cgroup adhere to the constraints imposed by the applied resource controllers.

Access control: The *principle of least privilege* mandates every entity to access only the resources that are necessary for its execution. Linux implements a set of mechanisms that assist processes to comply with this principle. For instance, the *Secure Computing (seccomp)* facility [Cor09, Cor12c] allows processes to register system call filters to limit their access to the considerable surface of the system call interface. Through seccomp, the kernel restricts processes in accessing the kernel's services by selectively reducing the number of authorized system calls. Specifically, containers can register and install seccomp filters in form of Berkeley Packet Filter (BPF) programs. This way, the in-kernel BPF bytecode Just-In-Time (JIT) compiler and interpreter can check the container's authorization of the requested service upon every system call invocation.

[3] The Open Container Initiative (OCI) specifies standards for the container runtime and image, yet, the implementation among different runtimes varies [Fou15b].

Additionally, Linux implements capabilities [Cor06, Edg15] and Linux Security Modules (LSM) [SFV20] that a container runtime can configure to enforce fine-grained access control policies. Since these, however, are deemed irrelevant for this work, we refer the reader to the respective documentation.

The Microsoft Windows kernel has not initially foreseen concepts similar to Linux namespaces and cgroups. Yet, to meet modern demands for OS-level virtualization and comply with the standards of OCI Microsoft added a subsystem for creating *silo objects* [YIRS17] to provide similar functionality. In addition, Microsoft implemented the *Host Compute Service (HCS)* [Mic17b], an abstraction layer that interfaces containers with underlying kernel subsystems without exposing their implementation details. That is, similar to Linux kernel sandboxing features, the HCS is responsible for encapsulating Windows containers in resource-governed environments. Yet, by using these mechanisms alone, Windows cannot execute containers which were build for Linux (Sec. 2.1.3).

The benefits of OS-level virtualization techniques heavily promote utilizing containers in private and industry sectors. Given this trend, one must consider the security implications that might arise in the presence of compromised containers. Generally, from the security perspective, the isolation capabilities of containers are regarded as critical. Specifically, this is due to the fact that containers share the host's OS kernel and merely abstract the view on its global resources; successful kernel exploits from inside a container have the potential to impair the security of other containers and even the host itself. There exist attack vectors that can assist adversaries in breaking out or disclosing (and even manipulating) sensitive information from inside of containers [Gra16, LLW+18, Mic20a].

Such threats partially stem from vulnerable container deployment and management components or insufficiently strong configuration policies, which in turn, result from a lack of expertise. (Note that we consider all configuration issues that impair the container's key properties including access control, and resource isolation and governance.) While this is a relevant topic, we consider it out of scope; it is the task of container runtime developers and vendors, e.g., to provide sufficient documentation and strong default configuration policies. Other sources of this threat can arise from potential vulnerabilities in system calls that can act as an open gate to the underlying OS kernel. Independent of the exact strategy of the adversary, once inside the kernel, she inevitably gains the power to take control over the system. This emphasizes the need for stronger isolation capabilities, which can be obtained through the concepts of system virtualization.

2.1.3 System-level Virtualization

System VMs implement the necessary means to form virtual execution environments for entire computer systems. Software executing inside these artificially encapsulated surroundings stay in constant believe of a false reality, namely that the capsule they reside in is the physical system. Given that the virtualization layer of system VMs provides full compatibility to physical systems, the virtual environment can host OSes, including the

associated user space components, and even nested instances of system VMs, which, in turn, can create their own virtual universe for software [PG74, SN05]. Given the power of system virtualization, the following provides insights on its capabilities that have shaped our vision of virtualization-supported dynamic analysis and OS architectures.

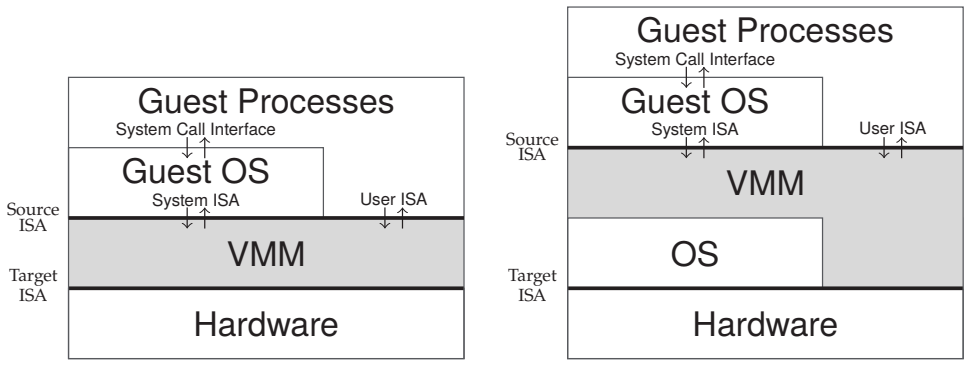
The software virtualization layer of system VMs was coined *virtual machine monitor (VMM)* in the 1960s, and is also referred to as the *hypervisor*.[4] Today, this layer is often assisted by hardware virtualization extensions. Unlike runtimes of process VMs, the VMM targets a level below the ABI and hence implements a virtual hardware interface. This interface comprises a virtual ISA, which, from the perspective of the software inside the VM, defines an abstract view of the *machine* (Fig. 2.1(a)). Similar to process VMs, the offered virtual ISA does not necessarily have to comply with the ISA provided by the underlying hardware. Such configurations resort to emulation techniques to translate the guest's instructions—as well as the entailed architectural characteristics—to the ones supported by the physical hardware. We have mentioned and referenced types of emulation techniques in Sec. 2.1.1. System VMs can apply these techniques to emulate legacy and experimental system environments for reasons of compatibility and research.

Further, the virtual ISA conceals the guest's perspective to physically available resources. Underneath the virtual ISA, the VMM maintains and manages the physically available hardware resources on behalf of different VMs. Each VM receives its own set of virtual hardware resources, which can comprise at least one CPU, memory, and I/O devices. In fact, the VMM takes on the complex task of transparently multiplexing the physically available, or software-emulated system resources through time-sharing or partitioning techniques, without disclosing potentially sensitive information of individual VMs. This way, the VMs receive the impression of exclusively owning the hardware, even though the VMM shares the available resources among them.

2.1.3.1 Hierarchical Privilege Separation

Modern computer architectures implement a strict privilege hierarchy that distributes privilege levels across different execution modes. Given the complex variety of hardware-defined privilege modes, we draw upon their logical abstraction from the OS's perspective: for simplicity, the OS distinguishes between the *user mode* and the *supervisor* or *kernel mode* (Sec. 2.2.1). In this regard, the high-level (user space) plumbing layer of the OS executes with user mode privileges; the low-level OS kernel space, in turn, operates with kernel mode privileges. (Note that concrete hardware architectures and OS implementations can assign these abstract modes to different physically-available modes [LSK18].) This constellation allows the OS kernel to maintain control over processes in user space. In other words, the privilege separation prohibits user space applications from accessing *privileged* resources. Examples of privileged resources are system configuration registers and the system's timer interrupt. Without enforcing an effective privilege separation,

[4] Throughout this work, we use the terms VMM and hypervisor interchangeably.



(a) Native VMM (Type I). (b) Hosted VMM (Type II).

Figure 2.4: VMMs execute on the host's (target) ISA and expose a virtual (source) ISA to guest OSes. While native VMMs (a) directly execute on bare-metal, hosted VMMs (b) operate as part of the OS [SN05].

(in the worst case) applications would be able to take indefinite control of the system's resources, and hence impair time-sharing and the concept of multiprogramming.

Having introduced the privilege hierarchy for modern OSes, we have to clarify how a VMM integrates and interacts with the rest of the system from an architectural perspective. Similar to how OSes impose control over their user space components, to maintain a certain level of control over VMs—or rather over the system's hardware resources—the VMM requires higher system privileges than the guest OS kernel in the VM. Consequently, the VMM should operate in a lower execution mode than the guest OS. The VMM's exact execution mode depends on its architecture. Generally, we distinguish between two types of system VM architectures, namely *native* (Type I) and *hosted* (Type II) system VMs [SN05]. Fig. 2.4 illustrates both flavours. Native system VMs operate directly on bare-metal, in one of the most privileged modes. The elevated privileges and the proximity to the hardware enable the VMM to be efficient and in direct control of the hardware. On the other hand, to accommodate a potentially high number of different architectures and their hardware configurations, native system VMs have to implement an arsenal of necessary drivers to govern and maintain the system's hardware resources. Hosted system VMs, on the other hand, execute as part of the host OS. In fact, the VMM of hosted VM architectures can reside entirely inside the OS's kernel, or, at the risk of reduced performance, outsource (at least) parts of its functionality into user space. Regardless, of the variation, hosted system VMs leverage the OS's services to communicate with the hardware, and hence benefit from a reduced complexity. In all cases, we use the term *system mode*, to refer to an abstract scope of jurisdiction of a VMM, disregarding its architecture and the concrete hardware-defined execution mode.

2.1.3.2 Efficient System VMs

To draw a more tangible picture of the term *privilege*, we borrow the taxonomy defined by the virtualization pioneers, Popek and Goldberg, who have summarized the formal requirements for system virtualization [PG74]. In the early 1970s, they have presented sufficient conditions for ISAs required to realize *efficient* system VMs. In this regard, they have defined a general classification for arbitrary ISAs, distributing instructions across the following three categories, namely *privileged*, *sensitive*, and *innocuous instructions*. Even though this long-established classification solely distinguishes between the user and supervisor mode, the concepts equally apply to the modes of modern architectures.

Privileged instructions: When executed in supervisor (or generally, in the most-privileged system mode) a privileged instruction does not trap (i.e., it does not cause the system to interrupt the instruction's execution and switch to a different mode with higher privileges). On the other hand, a privileged instruction must *always* trap, when it is executed in user mode.

Sensitive instructions: This class further distinguishes between *control-* and *behavior-sensitive* instructions. Control-sensitive instructions can modify the system's hardware configuration (e.g., they can register interrupt handlers). Behavior-sensitive instructions depend on the system's configuration and produce different results when executed, e.g., in different modes of operation.

Innocuous instructions: The set of instructions that does not belong to the group of sensitive instructions is considered innocuous.

According to the formal definition of Popek and Goldberg a system's ISA can be *efficiently* virtualized, only if the set of sensitive instructions is a subset of privileged instructions [PG74, SN05]. We elaborate on this definition below.

From a functional perspective, Popek and Goldberg describe the VMM as a *control program*, whose responsibilities are distributed across three modules, namely the *dispatcher*, *allocator*, and *interpreter*. The dispatcher handles the VM's implicit and explicit requests (i.e., VM exits). Explicit VM exits occur by explicitly generating *hypercalls* (e.g., traps generated by calling dedicated instructions). This mechanism is used to establish a direct communication between a VM and the VMM (e.g., to increase performance of I/O operations). Implicit VM exits occur every time the guest executes a privileged instruction, or upon hardware interrupts. In both cases, the VM traps into the VMM; this is where the dispatcher takes over control and delegates the incoming requests to one of the remaining modules. The task of the allocator is to maintain and safely multiplex the system's (physical and emulated) hardware resources across VMs. Finally, the interpreter assists the dispatcher in providing compatible functionality to emulate the traps into the VMM. Further, the work of Popek and Goldberg strictly specifies that a VMM is expected to satisfy the following three properties, namely *efficiency*, *resource control* and, *equivalence* [PG74].

Resource Control: This property requires the VMM to be in full control of the system's hardware resources. In other words, given this property, a VM cannot access the resources that were not explicitly allocated and granted to this VM in the first place. At the same time, this property ensures that the VMM is able to take back control over the resources granted to a VM.

Equivalence: The VMM is expected to establish a virtual execution environment that facilitates software to exhibit behavior equivalent to the behavior the software would show if it was executed on a physical machine. Tolerated exceptions to this property comprise the availability of physical resources and discrepancies in performance and timing characteristics.

Efficiency: Efficient VMMs allow VMs to execute all innocuous instructions directly on the CPU, without any interventions.

The efficiency property implies a given *efficiently* virtualizable ISA. Unfortunately, this assumption is not always supported by the underlying hardware architecture. For instance, the modern ISA, IA-32, alone does not satisfy the efficiency condition; IA-32 holds a set of sensitive, yet, not privileged instructions. Such *critical* instructions do not trap into the VMM, without the need for emulation (or paravirtualization) techniques [SN05]. (Naturally, the same argument applies to VMMs that expose a source ISA that is different to the target ISA of the physical system.) Yet, modern hardware-assisted *virtualization extensions* to various ISAs (including IA-32) address this issue, and hence facilitate implementing efficient system VMs. In this work, we rely on many features of the hardware-supported virtualization extensions. Thus, we discuss them in the following section.

2.2 Hardware-assisted Virtualization Extensions

Modern architectures often implement hardware extensions to support system virtualization (Sec. 2.1.3.2). Such hardware-supported virtualization extensions enhance the capabilities of the system's ISA to assist VMMs and guest OSes on processor level. From a birds-eye perspective, the CPUs equipped with virtualization extensions implement primitives that allow the VMM to provide guest OSes an abstract and isolated view on the physically available resources. Such extensions are specifically dedicated to reduce the virtualization-induced performance overhead and implementation complexity of the VMM, but also to establish a basis for efficient system VMs [PG74] (Sec. 2.1.3). In this section, we focus on the main system components that receive hardware support for system virtualization. These comprise the *CPU*, *memory*, and *devices*. We mainly consider the virtualization extensions of the ARM [Arm20] and x86 [Int20a] architecture, as they are relevant for this work. Yet, before diving into the details of the virtualization extensions, we take a step back to get a clear picture of the *execution environments* of both architectures. Note that we have published parts of this section in [PKZ18, PLM+18, PMG+20].

2.2.1 Execution Environments

We recall from Sec. 2.1.3 that modern computer architectures implement a strict privilege hierarchy, which allows high-privileged software to impose control over software with less privileges. The execution environments associated with the individual levels of the privilege hierarchy strongly depend on the hardware architecture and hence differ between x86 and ARM. Throughout this work, we mainly focus on the x86-64[5] [Int20a] and the two most recent versions of the A-profile (application) ARM architecture family, namely ARMv7-A [ARM14] and ARMv8-A [Arm20]. Also, because of their architectural resemblance, we use the ARMv8-A architecture to represent both ARMv7-A and ARMv8-A, unless stated otherwise. While the x86 architecture uses different *modes of operation* [Int20a], ARMv8-A refers to different *execution* and *security states* [Arm20] to describe the system's execution environments. In both architectures, the respective execution environments define the memory model, register set, and the set of available instructions.

2.2.1.1 Modes of Operation on x86-64

The modern x86-64 architecture comprises three main *modes of operation* (or simply *modes*) that define the ISA and hence form the execution environment for software components [Int20a]. These modes comprise the *real mode*, *protected mode*, and *IA-32e mode*—which is also known as the *long mode*. For brevity, we exclude the System Management Mode (SMM), a highly privileged mode for system monitoring and debugging. The real mode uses a 16-bit ISA that goes back to Intel 8086. The protected mode extends the memory

[5] We prefer Intel over AMD for describing the general architecture. Both architectures exhibit only negligible differences that are of no relevance for this section.

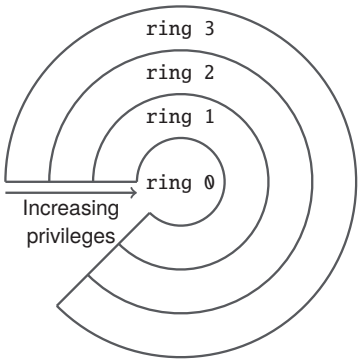


Figure 2.5: The privilege hierarchy on x86 implements four protection rings. Typically `ring 0` hosts the OS kernel, whereas user space processes execute in `ring 3` [Int20a].

protection capabilities of the real mode (through paging) and implements a backward-compatible mode for the 16-bit ISA and a 32-bit ISA that is known as IA-32. Finally, the IA-32e, or long mode, forms a 64-bit extension to IA-32. Similarly, to the protected mode, the long mode, comprises two sub-modes, namely the *compatibility mode* and the *64-bit mode*. The compatibility mode is binary compatible to the 32-bit IA-32 ISA, whereas the 64-bit mode implements the Intel 64 ISA (also known as AMD64).

Compatibility is an inherent characteristic of the x86 architecture; even today, x86-based systems begin their boot process in real mode before they can enter the protected or long mode. Once the system has switched to either one of these modes, it becomes able to logically organize different levels of the software architecture by assigning different privilege levels to individual code and data segments. The x86 privilege hierarchy comprises four privilege levels, numbered from 0 to 3 [Int20a]. This hierarchy uses a model, which associates every privilege level with a dedicated *privilege ring* (Fig. 2.5). Typically, the OS kernel operates in the most-privileged level, `ring 0`, whereas user space applications execute with the least privileges in `ring 3`. In other words, these two hardware-defined levels characterize the privileges of the introduced OS's abstractions for user and kernel mode (Sec. 2.1.3). If a user space process (in user mode) requires services of the OS kernel (in kernel mode), it must invoke a *system call* that temporarily suspends the process and performs the necessary actions to switch from `ring 3` to `ring 0`, in which the kernel can satisfy the request and return control to the suspended process in user mode.

2.2.1.2 Execution and Security States on ARMv8-A

The ARMv8 architecture differentiates between two *execution states*, namely *AArch32* and *AArch64* [Arm20]. The execution state AArch32 is binary compatible to the ARMv7-A architecture. It establishes a 32-bit execution environment that can be configured to host software compiled against the Thumb ISA, T32, and the ARM ISA, A32. The execution state AArch64, in turn, supplies a 64-bit ARM ISA, A64. Similar to x86, ARM implements

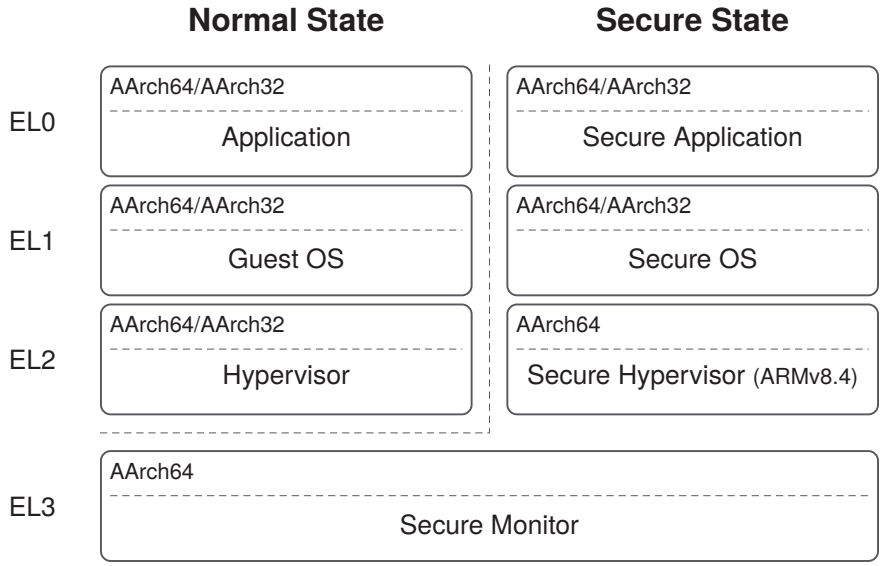


Figure 2.6: The privilege hierarchy on the ARMv8 architecture distributes privileges across different exception levels and security states [Arm20].

a privilege hierarchy, which assists the system to assign different levels of privileges to different software components. These hierarchical levels are called *privilege levels (PLs)* on AArch32 and *exception levels (ELs)* on AArch64. Both have only moderate differences, which are of no relevance to our work. To ensure a consistent terminology, we prefer the term exception levels over privilege levels. Also, we use the term ELs to refer to the individual levels of the privilege hierarchy of both execution states, AArch32 and AArch64.

Fig. 2.6 illustrates four different exception levels, numbered from 0 to 3 [Arm20]; higher-numbered ELs exhibit higher privileges. Also, individual ELs can use different execution states [Arm20]. For instance, components of the software architecture that execute in EL0 and EL1 can use AArch32, with software in EL2 executing under AArch64. Much like the protection rings on x86, the different ELs restrict access to privileged resources. This allows ARM to strictly separate and distribute individual components of the software architecture across different ELs. For instance, the OS kernel operates in the high-privileged EL1, whereas user space processes execute with the least privileges in EL0. The interaction between two different exception levels uses, as the name suggests, exceptions. Every exception that is taken from the currently active EL to the same or to a higher-privileged EL. Hence, every software component registers dedicated exception vectors which act as entry points of the individual ELs. For instance, every time a user space process invokes a system call, the system generates and takes the Supervisor Call exception from EL0 to the registered entry point in the exception vector table of EL1, where it resumes the operation.

As we can see in Fig. 2.6, ARMv8-A further distinguishes between the two different security states: *normal state* and *secure state*, with the latter being also referred to as *TrustZone*. The ARMv8-A architecture mirrors the exception levels EL0-1 (the ARMv8.4-A specifica-

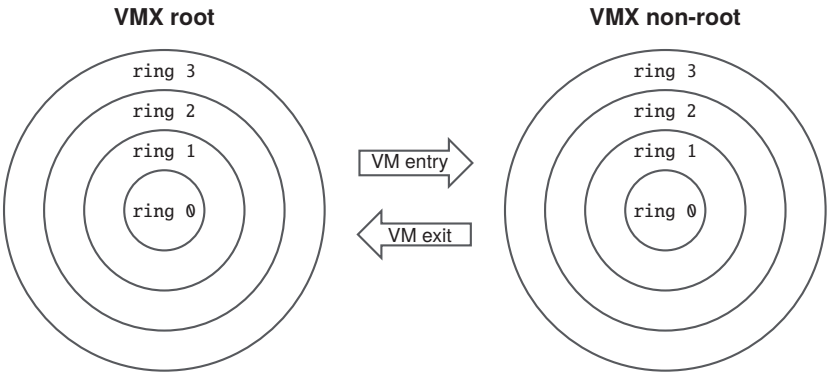


Figure 2.7: The x86 VMX mode of operations mirror the four protection rings [Int20a].

tion extends this architecture by also mirroring `EL2`) and places the copy into the secure state, which logically separates hardware resources in an attempt to establish a Trusted Execution Environment (TEE). The secure state additionally manages the `EL3`, which is mainly responsible for handling switches between the two security states.

2.2.2 CPU Virtualization

The essence behind CPU virtualization is to provide VMs with an illusion of having exclusive ownership of the CPU. To face this challenge, modern VMMs leverage hardware-supported virtualization extensions to temporarily assign and multiplex the system's CPU among the VMs, without losing control of the CPU to either one of them. To ensure that the VMM maintains control over the CPU, the virtualization extensions force each VM to operate on its own, virtual state; the hardware ensures that the privileged instructions trap into the VMM, in case they would otherwise affect the state of other VMs or the VMM.

Different hardware implementations of virtualization extensions facilitate CPU virtualization by dedicating an additional, high-privileged execution environment to the VMM. Having discussed the concepts behind various execution environments on x86 and ARM in the previous section, we can turn our attention towards the specifics of the virtualization extensions of these two architectures. Therefore, the following summarizes the concepts applied by the virtualization extensions on Intel and ARM to virtualize the CPU.

2.2.2.1 Intel Virtualization Technology

Intel's virtualization extensions (Intel VT-x) introduce an additional mode of operation, the virtual machine extensions (VMX) operation [Int20a]. This mode differentiates between two logically separated sub-modes comprising the *VMX root* and *VMX non-root* operation. The VMX root operation extends the system's ISA with an additional set of VMX instructions to maintain virtual environments. At the same time, VMX non-root imposes restrictions on the software privileges, behavior, and perspective on the underlying

system. For instance, selected hardware events or privileged instructions cause software in VMX non-root to trap into the VMX root operation. Typically, a VMM operates in the high-privileged VMX root and guest VMs operate in the less-privileged VMX non-root.

To provide full compatibility for software architectures in both the VMX root and VMX non-root, Intel VT-x duplicates the four privilege levels and assigns both sub-modes of the VMX operation their own set of protection rings (Fig. 2.7). In this way, the software architecture of, e.g., OSes does not have to be adjusted for the respective VMX operation. Intel VT-x dedicates a hardware defined data structure called *Virtual Machine Control Structure (VMCS)* to maintain control over the VMs in VMX non-root and coordinate transitions between the VMM and a particular guest. Specifically, the VMCS reserves space for the host's CPU and the guest's virtual CPU (vCPU) state, which is used to temporarily save and restore the particular state on every VMX transition. That is, the system initializes the CPU with the guest's vCPU state on VM entries and restores the host's CPU state on VM exits.[6] If the guest requires multiple CPUs, the VMM must maintain one VMCS for each virtual CPU. Further organization of the VMCS holds execution control fields that determine the guest's behavior. For instance, by configuring these fields the VMM can define the set of events that will trap into the VMM and the way how the system behaves on VM entries and exits.

2.2.2.2 ARM Virtualization Technology

ARM offers its own set of virtualization extensions to assist CPU virtualization [Arm20]. Similar to Intel VT-x, the virtualization extensions of the ARMv8-A architecture dedicates the high–privileged exception level, `EL2`, for hosting a VMM (Fig. 2.6). `EL2` equips the VMM with capabilities to maintain and impose control over VMs. Software components in `EL0` or `EL1` trap into the higher privileged VMM in `EL2`, e.g., on privileged instruction fetches, or on access to system configuration registers or devices managed by the VMM. Similar to system call invocations that take the generated Supervisor Call exception from `EL0` to `EL1`, the VMM initializes a dedicated exception vector table that allows to handle exceptions from either of these exception levels. Thus, a VM exit on ARM is nothing else than an exception taken to `EL2`. Contrary to the VMCS of Intel VT-x, the virtualization extensions on ARMv8-A do not automatically save and restore the state of guest OSes from a hardware-defined memory region. Instead, ARM implements different banks for special- and general-purpose registers. These banks bind selected registers to a particular execution environment, and preserve their contents in their banked shadow copies on transitions to another exception level or execution and security state. Additionally, the VMM maintains the respective unbanked CPU state on VM transitions. Even though this design decision entrusts the software with managing the CPU state, the VMM can focus on saving only relevant information, thus avoiding unnecessary memory writes.

[6] The terms *VM entry* and *VM exit* refer to Intel's terminology describing transitions from the VMM into the guest and reverse. Throughout this work, we use these terms to also describe the transitions between the VMs and the VMM on ARM systems.

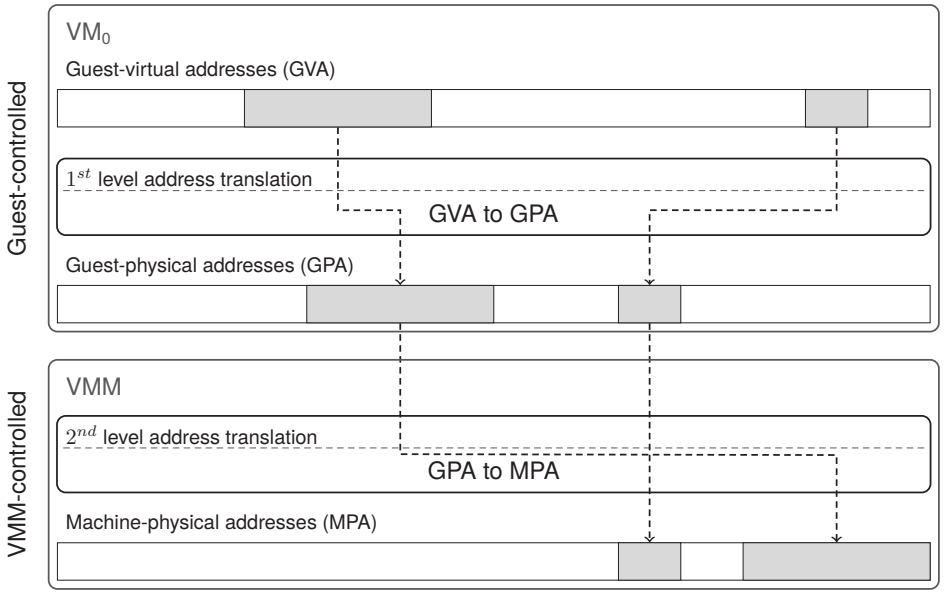


Figure 2.8: The system's virtual memory system architecture uses guest-controlled first level address translation and VMM-controlled second-level address table translation tables to translate guest-virtual to machine-physical addresses.

2.2.3 Memory Virtualization

Through memory virtualization, VMMs provide the illusion to VMs of having control of their physical memory. While guests maintain page tables for translating guest-virtual to guest-physical addresses, the VMM is responsible for translating guest-physical into machine-physical addresses (Fig. 2.8). This lends VMMs a memory isolation property that ensures that even compromised guests cannot manipulate other VMs or the VMM itself. On systems without virtualization support, the VMM maintains an additional set of page tables (*shadow page tables*) for managing the guest's physical memory in software. By write-protecting the guest's page tables, the VMM intercepts their modifications and redirects translations to a dedicated memory location. The downside is that the complex software management of shadow page tables entails a significant performance overhead.

To approach the poor performance of shadow page tables on hardware with virtualization support, the Memory Management Unit (MMU) features a supplementary level of indirection through the *second level address translation (SLAT)* tables (Fig. 2.8). Similar to shadow page tables, the hardware requires an additional set of page tables that is maintained solely by the VMM and cannot be accessed by the guest. These second stage translation tables complement the translation of guest-virtual to guest-physical addresses, by mapping the guest-physical to machine-physical addresses. Accesses to memory that is not mapped or lacks access permissions in these tables result in traps allowing the VMM to isolate and control the guest's view on the physical memory. Different hardware implementations of SLAT tables can differ. For instance, the SLAT tables of both AArch64

and Intel allow to define *execute-only* memory, which lends a VMM the ability to hide code instrumentation [DZX13, LMP+14, PLM+18]. In contrast to Intel and AArch64, SLAT on AArch32 does not support this functionality.

2.2.4 Device Virtualization

Similar to the system's CPU or memory, Input/Output (I/O) devices describe yet another form of system resources governed by the VMM. The versatility of various devices strongly affects the task of device virtualization. In contrast to a CPU, which conforms to a single interface (i.e., ISA), devices implement a high number of different interfaces, which have to be considered by the VMM. Generally, a VMM equips VMs with a set of virtual devices. Virtual devices can, but do not necessarily have to correspond to the physically available devices. In cases, in which virtual devices are not backed by their physical representation, the VMM can intercept the VM's I/O requests and emulate the device's behavior in software. Regardless of the device's implementation, the VMM bears a two-fold responsibility to virtualize devices. First, the VMM has to ensure a compatible device representation towards VMs. This task differentiates between shared and non-shared devices. In particular, the VMM must determine how to correctly manage the device's state. This presents a challenge for devices, which were not implemented with virtualization in mind. For instance, while a VMM can divide and assign individual partitions of a hard drive to different VMs, multiplexing access to network devices or graphics cards among different VMs dramatically increases the management overhead. Second, the VMM is responsible for providing a communication interface between the VMs and devices. To satisfy this task, the VMM can: (i) trap and emulate every I/O request in software; (ii) modify drivers of the guest OS to establish an efficient, virtualization-aware communication; or (iii) employ hardware virtualization techniques. Modern architectures typically apply para- or hardware-virtualization techniques to virtualize devices. When applying para-virtualization, the system leverages a split-driver architecture [Chi08], in which a device driver's functionality is split across a VMM-aware front end (inside the guest) and a back end (inside a privileged VM or the hypervisor). Both driver components communicate through a well-defined interface (e.g., `virtio` [Rus08]). In contrast, hardware-virtualization directly passes through a physically available device to a particular VM. Such techniques are often accompanied by an I/O Memory Management Unit (IOMMU) [Int19], which allows to remap and isolate the address spaces of DMA-capable devices [BYMX+06]. Sophisticated virtualization extensions [Int20c] additionally manage the devices' state, and hence multiplex devices among different VMs. In this way, hardware-supported device virtualization reduces the complexity and interactions with the VMM and hence increases the overall performance.

2.3 Virtual Machine Introspection

Many modern OSes bundle the most sensitive and security relevant components at its heart, namely in the depths of the OS kernel. Yet, contrary to microkernel architectures, a monolithic design does not foresee a privilege separation layer between the security-related and the remaining kernel components, or rather subsystems. Even though the long-established concept of isolation has proven effective in increasing security [SS72], unfortunately, logically separating kernel subsystems is not the most relevant point on the kernel developers' agenda. At the same time, this lack of isolation encourages maliciously motivated actors to target and abuse vulnerabilities in various interfaces of the OS to establish primitives to illegally access the memory reserved for the kernel. These primitives can allow attackers to read from or write to sensitive data in the kernel memory, or cause the kernel to delude or even deactivate the given security mechanisms. Sophisticated malware can counter defense mechanisms by evading or directly attacking trusted pillars upon which the defenses rely [JBZ+14]. To approach this issue, Chen et al. [CN01] suggested to relocate OSes into VMs and to shift the OS's security services from the OS kernel into an isolated and high-privileged environment that is maintained by a VMM. The main reason for this is that security applications, which operate outside the VM cannot be easily manipulated or fooled by a potentially compromised OS. With their work, Chen et al. have set an incentive to leverage virtualization technology to support OSes, which, shortly after, emerged to a novel research direction, coined virtual machine introspection (VMI) [GR03]. VMI has highly influenced our work and is the main focus of this section.

2.3.1 The Scope and Benefits

Virtual machine introspection describes primitives that facilitate software to engage in OS hardening and analysis activities from an isolated, and hence protected environment. Specifically, VMI comprises a set of techniques that utilize the VMM to inspect, manipulate, and control software executing inside VMs [Pfo13]. These capabilities lend VMI applications an omniscient character, which is in particular attractive for its unique property of having a complete and untainted view over the VM's state [PG74, Pfo13]. These properties have proven effective and led to an increased adoption of virtualization techniques for malware detection, prevention, and analysis frameworks in commercial [VMR20, Fir20, Bit20] and open source sectors [VKSE13, LMP+14, Lib20, PSE11].

The concepts behind virtual machine introspection entrust a VMM to regain the trust in security that is increasingly questioned in non-virtualized OSes [JBZ+14]. The fading lack of trust in OSes can be attributed to their constantly growing size and complexity; both properties inadvertently introduce sufficient ground for malicious actors to invade and take over the kernel. In contrast, VMMs are limited in size and complexity. In addition, they expose only a manageable-sized interface towards VMs, isolate themselves from malicious activity, and hence render themselves as suitable candidates for the system's Trusted Computing Base (TCB). Consequently, shifting security applications out of the OS

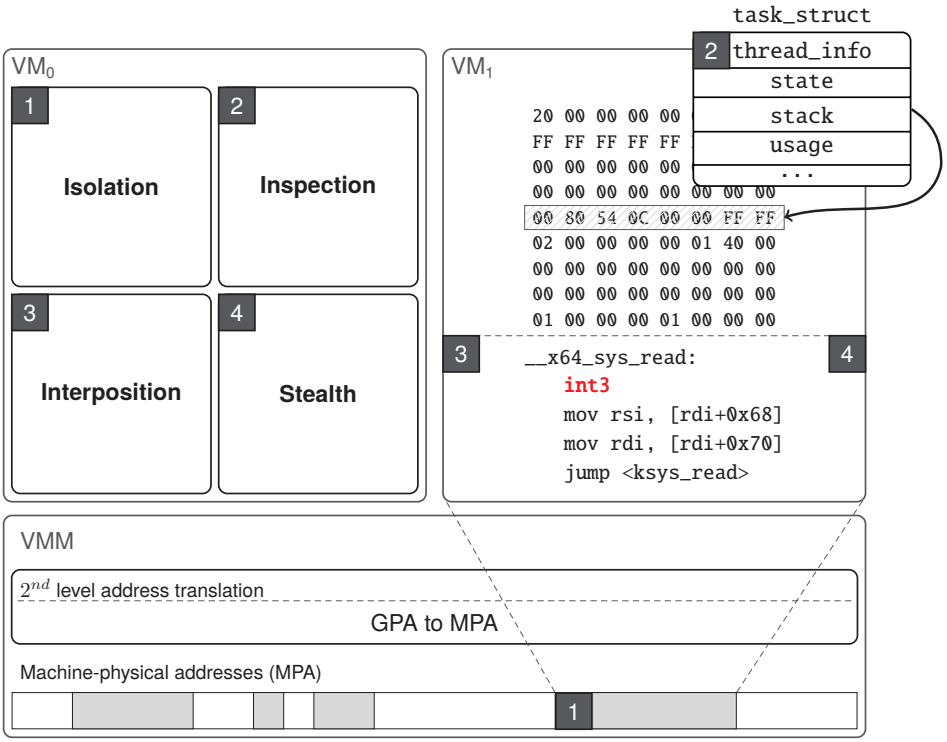


Figure 2.9: A VMM must supply the VM_0 with the capabilities *isolation*, *inspection*, *interposition*, and *stealth* to facilitate stealthy introspection of the guest OS inside VM_1.

into the VMM is a logically-appealing strategy. Garfinkel et al. [GR03] have summarized three capabilities a VMM must supply to accommodate the necessary foundation for VMI. These capabilities comprise *isolation*, *inspection*, and *interposition*. In addition, recent research [DZX13, ZLSS13, ZLS+15, LMP+14, Vog15, Len15, NZ17] has identified a fourth capability that has become indispensable for VMI-based malware analysis, namely *stealth* (Fig. 2.9). In the following, we summarize the four core characteristics a VMM has to supply to support stealthy virtual machine introspection.

Isolation: The inherent characteristics of VMMs provide guest software executing inside a VM an illusive perspective on the system's resources (Sec. 2.1.3); guests cannot access resources that were not explicitly granted and are virtually unaware of any environment outside the virtual machine. These characteristics form the isolation property, which ensures that even potentially compromised guest OSes cannot affect the state of other VMs or the VMM itself [GR03]. Consequently, from a security standpoint, the isolation capability ensures the integrity of VMI-based security mechanisms executing outside of the target VMs. Fig. 2.9 provides a simplified representation of the isolation property (❶). The VMM utilizes the second level address translation to isolate the memory of individual VMs. Even though, the VMM supplies further capabilities to isolate the system's resources, we exclude them for the sake of brevity.

Inspection: Generally, malware either aims to directly attack or evade security mechanisms, e.g., by feeding them with falsified information [GR03]. The strong isolation capability of VMMs ideally thwarts direct attacks against outsourced VMI components. On the other hand, the inspection capability can assist VMI to obstruct evasion attacks. This capability hinges upon a VMM with a complete and untainted view of the guest's memory and register state, as well as the state of virtual devices [GR03, Pfo13]. Equipped with these insights, VMI applications cannot be easily deluded. Yet, the VM's state involves a great amount of binary information that VMI applications need to put into the right context, before they can apply any security-relevant actions. To translate the VM's binary state to high-level information, VMI applications have to trust and rely on semantic information of the guest's OS and the virtual hardware architecture, which has to be gathered in the first place [PSE09]. For instance, the VM_0 in Fig. 2.9 applies semantic information associated with VM_1 to map its binary state to the high-level data structure `struct task_struct` (❷). Generally, this lack of knowledge is known as the *semantic gap*, which every VMI application has to bridge [CN01]. Because of its relevance for VMI, we pay special attention to the semantic gap in the following section.

Interposition: To avoid losing control over the system's resources to guest OSes, the VMM must provide the necessary means to interpose, or intercept the guest's operation upon certain events. Such events can comprise, amongst others, the guest's execution attempts of privileged instructions, access to protected memory regions, and hardware interrupts (Sec. 2.1.3). While some of these events unavoidably trap to temporarily hand over control to the VMM for managing resources, the system can be configured to assist VMI; intercepting selected events of interest lends VMI tools the ability to observe and manipulate, and hence control the target VM's state for various security purposes. For instance, a VMI application can utilize the VMM to write-protect the memory region of the guest's Interrupt Descriptor Table (IDT) to receive notifications about every unauthorized in-guest write access to it.

Despite the VMM's power to intercept the VM's events, its applicability reaches its limits when protecting the security sensitive resources that are not bound to the hardware. In other words, the hardware does not foresee the necessary means to intercept certain events, even though they are relevant for the purpose of analysis; occurrences of such events do not trap into the VMM. Thus, often, VMI tools must apply alternative and creative methods to side-step the hardware's inability to intercept events of interest to nevertheless achieve the desired behavior. For instance, even though the VMM cannot directly intercept system calls on x86, Pfoh et al. [PSE11] have shown an alternative approach: the authors utilize their VMI framework, Nitro, to initialize the Model Specific Register (MSR) holding the base address of the system call dispatcher with `NULL`. Alternatively, invasive introspection frameworks, such as DRAKVUF [LMP+14, PLM+18], inject, e.g., software breakpoint instructions into the guest's memory to interpose on selected events. Fig. 2.9 illustrates a similar setup

in which we inject an `int3` instruction into the prologue of the `__x64_sys_read()` system call handler (❸). This configuration causes the system to trap into the VMM on every system call invocation. As such, intercepting events of interest can impose a measurable performance overhead, introducing another source of information for malware that aims to evade virtual environments [Rut04, CAM+08, MANP17].

Stealth: Malware can exhibit a split personality. In other words, malware can change its behavior, as soon as it uncovers suspicious artifacts, or merely suspects an underlying analysis framework. Instances of this split-personality malware employ anti-debugging and anti-virtualization techniques [Rut04, CAM+08, BCK+10, SAM14, BBF+16, MANP17] to reveal and evade analysis. Similar to the *isolation* capability, *stealth* is not to be confused with *VM transparency* [GAWF07]. One can approach VM transparency in an attempt to make the VM indistinguishable from real hardware to reduce virtualization artifacts (that present themselves in form of discrepancies to physical machines in the execution environment, hardware, and system behavior), and hence avoid malware evasion; yet, according to Garfinkel et al., and to this date, this property remains infeasible, and more importantly impractical for modern defense mechanisms [GAWF07]. Disregarding the technical challenges, which make this goal infeasible in the first place, its impracticality is mainly because of the fact that, today, we observe an increasing trend toward system consolidation through virtualization. This trend renders the goal of *VM transparency* obsolete; the presence of a virtualized environment does not necessarily indicate that its sole purpose is dedicated for VMI [PLM+18]. Therefore, it would be unprofitable for attackers to exclude virtual environments. Even though perfect VM transparency is still infeasible [GAWF07], as we point out in Fig. 2.9, VMI tools should focus on hiding *analysis artifacts*, such as memory or register contents that can indicate and, in the worst case, reveal the underlying analysis framework (❹). By providing the necessary means to cloak analysis artifacts, a VMM can facilitate stealthy monitoring of highly sophisticated malware.

A VMM that inhibits this (extended) set of capabilities supplies VMI-based security mechanisms with a proficiency specifically tailored for efficient and highly robust malware detection and analysis, conducted entirely isolated from the software inside the VM.

2.3.2 The Semantic Gap

Modern OS kernels comprise various data structures, each dedicated for a specific purpose. These data structures hold small fragments, or snapshots of the entire system's state and assist the OS in managing and maintaining its resources. In other words, OS kernel data structures define building blocks required to form high-level abstractions, such as processes, files, and kernel modules. High-level abstractions convey the OS's semantics that fully describe the state of the OS. For instance, monitoring tools, such as `ps` on Linux, observe the OS's data structures to infer semantic knowledge about the respective high-level abstractions (i.e., in the case of `ps`, the semantic knowledge describes the state information of active processes). Consequently, it is this semantic knowledge upon which many security applications rely to detect and potentially prevent malicious activity.

From the VMM's perspective, the state of a VM comprises a vast amount of unstructured binary information in volatile memory and registers, as well as disk contents and devices.[7] The lack of structure underneath the OS obfuscates a correct interpretation of the semantics. Even though the VMM provides the necessary means to interpose on various in-guest events and let VMI tools observe the complete and unobstructed binary state of the VM, without first placing this binary information into the right context, the VMM, and hence the VMI applications, will not be able to make sense out of it. We refer to this lack of knowledge as the *semantic gap* [CN01]. Consequently, every VMI application faces the challenge of interpreting and reconstructing the hidden high-level semantics from the immense amount of low-level information to *bridge* the semantic gap [PSE09].

2.3.2.1 Bridging the Semantic Gap

The general mechanism behind bridging the semantic gap is to apply information, or knowledge that facilitates generating a semantically enriched *view* on the VM's binary state. The added information constitutes in-depth knowledge of the guest OS (i.e., the software architecture) and (virtual) hardware architecture that assists VMI tools in reconstructing data structures to regain fragments of the missing semantics. Recent research has organized common methods for reconstructing OS kernel data structures into three *view generation patterns* [PSE09]. These comprise the *in-band delivery*, *out-of-band delivery*, and the *derivative* pattern, which we briefly outline in the following. Depending on the scenario, these patterns can be applied separately or in combination to achieve most effective results. They apply different methodologies to acquire the knowledge required to bridge the semantic gap. More importantly, the applied methodologies differ, among others, in completeness of the reconstructed state, stealth, and their assumptions concerning trust in the guest OS [Pfo13, JBZ+14, Vog15].

[7] Throughout this work, we reduce the notion of the VM's state to the virtual architecture's register state and contents of the volatile memory.

In-band delivery pattern: This pattern involves in-guest agents that utilize services of the guest OS itself to acquire the state information of interest and deliver it to the VMM. The gathered state solely considers the guest's software architecture and is thus delivered in-band (i.e., from within the VM). The in-band delivery strategy benefits from its unique position, as it can completely avoid the complex and potentially inaccurate operation of bridging the semantic gap; the agent does not need to reconstruct the guest's data structures as it already has insight to the guest-visible state. That is, in-guest agents observe the guest's state in a similar level of detail as it is perceived by the guest OS. Therefore, the in-guest agent rather observes and passes the first-hand knowledge about the guest's software architecture to the VMM.

At the same time, such agents expose themselves to the in-guest environment and are subject to potential attacks. Even though a VMM can shield the agent against direct tampering attempts, it cannot guarantee that the agent receives a complete and unobstructed view of the guest's state; malware can detect such agents and feed them with falsified information. The question of trust applies similarly to the guest OS as the agent strongly relies on its trustworthiness; a compromised OS can break the inflicted assumptions of its benignity, and hence similarly mislead the in-guest agent. Therefore, in-band delivery patterns trade off stealth and completeness for convenience and simplicity of inspecting and delivering the state information. From a critical perspective, the given downside questions the benefits of approaches utilizing an in-band delivery strategy over traditional in-guest security measures, such as common off-the-shelf anti-virus solutions.

Out-of-band delivery pattern: This pattern involves an external component to identify and extract information necessary to reconstruct guest OS kernel data structures. This component is responsible to supply the VMI framework with the accumulated knowledge. Empowered with this knowledge, the VMI framework becomes able to transform the guest's binary representation into a meaningful state that is necessary to derive the high-level semantic insights. Only then, the VMI framework can start with the guest introspection activities. Generally, such external, knowledge-gathering components acquire the semantic knowledge ahead of time (i.e., before the VMI operation). This knowledge can comprise exported or debugging symbols, crafted data structure signatures, or the results of static or dynamic analysis that can help to narrow down the format and location of relevant kernel data structures in the guest's memory [JBZ+14].

Contrary to the in-band delivery pattern, strategies that acquire the relevant information out-of-band do not engage in any in-guest participation, and hence do not risk a direct exposure of the analysis framework. Thus, out-of-band delivery approaches are suited for stealthy operation. On the other hand, the view generation component of the VMI framework strongly relies on the trustworthiness and competency of the external component that delivers the acquired semantic knowledge

in the first place. The delivered information is as good as the applied information gathering technique and likely to not cover the guest's complete state. Another level of uncertainty applies to the integrity of the guest OS; the VMI framework cannot be completely sure that the delivered semantic knowledge fully matches the guest OS's software architecture. This is because the delivered semantic knowledge cannot be bound to the respective OS architecture and hence cannot be trusted [Lit08, JBZ+14]. Even if the OS was benign, malicious actors could inflict, e.g., unauthorized structural changes on the OS kernel data structures to confuse the VMI framework and evade analysis [BJW+10].

Derivative pattern: Contrary to the information delivery patterns, this pattern uses in-depth knowledge of the (virtual) hardware architecture alone to derive a semantically enriched view on the guest's state. One of the biggest advantages of this pattern is that the derived information does not depend on any software components that are (potentially) susceptible to deception. Instead, this strategy trusts and binds its assumptions to the underlying hardware specification, rendering them as immutable; the entire software architecture inside the VM adheres to this specification. Consequently, even the most sophisticated malware cannot affect the assumptions, which were made to derive the view. For instance, OSes leverage hardware-defined data structures (such as segment and interrupt descriptors, and page tables) as anchors (locked, e.g., in the system's control registers) that define the environment the guest OS relies upon; malicious actors cannot simply neglect or modify these anchors and hence similarly depend on their existence. Hardware vendors precisely describe the structure of such hardware anchors and (parts of) the referenced data structures. Consequently, derivative patterns can uncover parts of the guest's state by reconstructing data structures along the chain of interlinked references stemming from the hardware anchor. In other words, data structures are *rooted in hardware* if it is possible to create a chain of references between a hardware anchor and the particular data structure [Pfo13].

At the same time, purely derivative patterns reconstruct only a limited view on the guest's state. Only a small number of data structures inside OSes are *rooted* in hardware [Pfo13]. As such, they are often employed complementary with the introduced information delivery patterns.

The given view generation patterns equip external VMI tools with the ability to reconstruct a part of the guest's state to establish robust security frameworks. At the same time, they place strong trust assumptions on the integrity and benignity of the component, which is responsible to gather the semantic knowledge, and on the OS itself. We consider this an open problem and hence dedicate the following section to clarify the issue at hand.

2.3.2.2 The Question of Trust

VMI tools apply one, or a combination of the presented view generation patterns to translate the VM's binary representation into meaningful information (Sec. 2.3.2.1). In this context, VMI tools, which employ in-band or out-of-band delivery patterns, assume trustworthiness and integrity of the following three components: (i) the component that gathers the semantic knowledge, (ii) the delivered semantic knowledge itself, and (iii) the guest OS kernel [JBZ+14]. The trust assumptions placed on these components cannot always be relied upon. The quintessence is that the delivered semantic knowledge is *non-binding* [Lit08]; it cannot be bound to an immutable anchor which is guaranteed to remain unchanged at run-time [JBZ+14]. By changing the assumptions of the OS's semantics, malware can evade introspection. For instance Direct Kernel Structure Manipulation (DKSM) [BJW+10] attacks specifically aim to modify the guest kernel's data structure representation. Disregarding their complexity, and whether such attacks are at all feasible in realistic scenarios, the net effect of DKSM attacks is that VMI tools continue to rely on outdated information and hence can potentially miss the malicious behavior. Jain et al. [JBZ+14] use these trust assumptions to differentiate between the *weak* and the *strong semantic gap* problem.

Weak semantic gap: This problem fully relies on the integrity of the acquired non-binding semantic knowledge. The gathered knowledge is immutable and complete in regard to all security-relevant invariants. Further, this problem trusts that the guest OS remains benign during the view generation and cannot be compromised until its deployment and VMI-controlled launch.

Strong semantic gap: This problem neither trusts the guest OS nor any non-binding semantic information without additional run-time validation [JBZ+14].

Under the assumption of the strong semantic gap problem, a maliciously driven semantic knowledge gathering component (whether in-guest or external) can falsify, withhold, or simply not identify potentially security-sensitive information during the acquisition of the semantic knowledge. Alternatively, a malicious actor can retrospectively modify the delivered semantic knowledge, without influencing the process of acquisition. In both cases, the modified or incomplete knowledge can create a blind spot in the reconstructed guest state, which would guarantee a safe space for malicious activity. Finally, even if (i) and (ii) were complete and trustworthy, the guest kernel itself (iii) can break the assumptions of VMI, e.g., if it was replaced or dynamically patched, right after having applied the delivery-based view generation patterns. On the other hand, derivative view generation patterns cannot be easily fooled. Yet, as we have learned in the previous section, purely derivative approaches cannot reconstruct the entire state of a guest. Hence, they should be accompanied by knowledge acquired either through in-band or out-of-band delivery patterns. Since not every data structure is rooted in hardware, the complementary delivered semantic knowledge faces the described challenges. As a consequence, the strong semantic gap problem remains unsolved and hence requires further investigation.

2.4 Summary

In this chapter, we have outlined the technical foundation that is required to assist the reader in better understanding the components of our work. The following chapters will draw upon the depicted foundation and further extend it where necessary. In essence, we have guided the reader through the fundamental concepts of virtualization technology, for virtualization technology has inspired and is the main topic of this work. In an attempt to share both our insights and admiration to virtualization technology, we progress through different forms of virtualization that are common today. Further, we amplify upon selected hardware components and extensions of the x86 and ARM architecture, which simplify system virtualization. Throughout this work, we repurpose the described hardware components in different ways to assist VM introspection as well as the security of OSes. We conclude this chapter with an overview of virtual machine introspection techniques, which entail advantages and disadvantages, when compared to conventional in-kernel security mechanisms. The main intention behind this chapter is to prime the reader such that they develop a sense for the high potential of virtualization technology in regard to OS security; the strong logical separation capabilities make virtualization techniques particularly attractive with regard to security, and hence to our work. Having defined the necessary foundation, we have set the sail towards exploring novel concepts that leverage virtualization for *stealthy* dynamic binary analysis, as well as virtualization-assisted OS security primitives (Sec. 1.2). Yet, before we dive into the details of our work, in the next chapter, we provide the reader with a conceptional system architecture, which combines the contributions of this work to convey our vision of a virtualization-assisted system architecture.

Chapter 3

System Architecture

Yes, three is mystic. Three stands at the heart of your quest.
Another number comes later. Now the number is three.
— Stephen King

This work investigates novel virtualization-assisted primitives to improve the state-of-the-art stealthy dynamic binary analysis techniques and the security of selected OS components (Sec. 1.2). To put the perspective on these two main drivers of our research into a more concrete frame, in this chapter, we introduce a conceptional and hardware-independent high-level representation of a system architecture. Specifically, the target system architecture in this chapter integrates the primitives introduced in this work and provides an organized overview of their concepts. In this regard, we define a target system hosting a VMM, which maintains two logically separated VMs—one for each research direction—and aligns our primitives with their scope of application in the respective VM. This allows us to position and shed light on the introduced security elements in the context of a global system representation and hence exemplify the purpose of our work. In addition, we outline a comprehensive threat model targeting the introduced system architecture. The threat model serves as a basis and is further extended by the elaborated threat models in the following chapters specifically tailored to the individual primitives.

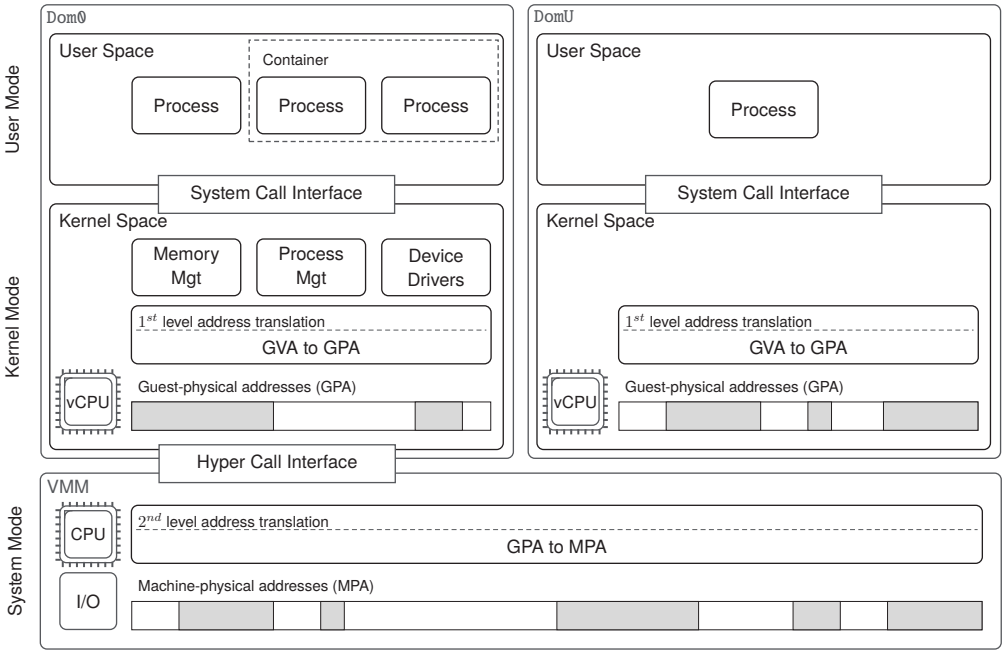


Figure 3.1: The target system architecture comprises a VMM and two domains, the privileged domain, `Dom0`, and the unprivileged domain, `DomU`. The domains represent and accommodate the two research directions of this work, namely stealthy dynamic binary analysis (`DomU`) and virtualization-assisted OS security (`Dom0`).

3.1 Conceptional Target System

The target system architecture serves as an abstract organizational structure. It assists the reader in aligning the objectives of this work with the different building blocks of the following chapters in form of virtualization-assisted security primitives. Even though our contributions build upon physical hardware capabilities, the conceptional system in this section generalizes the applied concepts, and hence avoids any hardware dependencies. Thus, the target system is not bound to any specific hardware architecture. Instead, it represents a wide-ranging set of architectures with hardware virtualization extensions. Real-world systems that can benefit from the introduced concepts vary from low-power mobile, IoT, and wearable devices, to legacy and high-end desktops, laptops, and servers. Most of the presented concepts are orthogonal to each other: while a combined setting exhibits the most benefits, the introduced security primitives can be applied individually.

Fig. 3.1 illustrates a simplified target system architecture comprising a VMM and two logically separated domains (i.e., VMs),[1] with each hosting a fully-fledged OS kernel and user space plumbing. Each domain represents one of this work's goals, in the context of which we leverage virtualization for (i) stealthy dynamic binary analysis and (ii) OS security (Sec. 1.2). Thus, the logical separation between both domains in the figure allows us to

[1] We prefer the Xen Project hypervisor's [BDF+03, Chi08, Lin20a] notion of *domains* for describing VMs.

clearly separate the two perspectives on security that guide our research, and to highlight how our contributions interact with the target system architecture to accommodate them. In this regard, we devote the unprivileged domain, `DomU`, on the right hand side of the figure to dynamic binary analysis. Specifically, the VMM of our target system architecture supplies a VMI-based framework in the privileged domain, `Dom0`, with access to novel primitives of the VMM to observe the behavior of binaries inside the `DomU` in a stealthy way. On the other hand, we use the `Dom0` on the left hand side of the figure to highlight the involved OS components, which we have equipped with virtualization-assisted memory protection primitives to increase the system's resilience against potential attacks. We elaborate the introduced primitives in detail in the following chapters (Chap. 4 – 7).

We distribute the virtualization-assisted primitives (which form the essence of this work) across different abstraction layers of the target system architecture. The following outlines the assumptions and responsibilities of involved abstraction layers and links them to the security primitives of this work. To maintain consistency, we adhere to the introduced terminology (Sec. 2.1.3), which associates the different abstraction layers of the architecture with execution modes (i.e., *user*, *kernel*, and *system mode*) of the privilege hierarchy.

System mode: The system mode hosts the VMM and forms the lowest and most-privileged level that we consider in the target system architecture. It supplies the VMM with services of the hardware-assisted virtualization extensions required to establish the foundation for (efficient) system VMs (Sec. 2.1.3). We assume that the target architecture supplies a SLAT mechanism to isolate and govern access to physical memory that is reserved for VMs (Sec. 2.2.3). Optionally, the system can encrypt the guest's memory, yet, this would limit the VMM's *inspection* capability and hence impair its functionality required to dynamically analyze binaries in the `DomU` (Sec. 2.3.1).

Aligning the contributions: The VMM operates with privileges of the system mode and is the central component that is responsible for implementing the security primitives for both stealthy dynamic binary analysis and virtualization-assisted OS security. We utilize and extend the Xen Project hypervisor [BDF+03, Chi08, Lin20a], yet, we do not limit the general concepts to Xen (Chap. 5 – 6). Alternatively, we propose using a thin VMM, which benefits from a reduced attack surface; it implements only essential functionality and can equip arbitrary virtualization-assisted primitives with an on-demand deployment strategy (Chap. 4). Regardless of the security primitives' deployment details, instead of using one set of SLAT tables to define a *global view* on the guest-physical memory, we assume the VMM has the ability to maintain and switch among *different views*. This is a requirement to form primitives to enable stealthy dynamic binary analysis (Chap. 5) and enhance the guests with strong memory isolation capabilities (Chap. 6). In both cases, the VMM offers authorized guest OS components access to these primitives through its hypercall interface. This way, guest OSes can leverage these primitives to stealthy analyze binaries in other VMs (Chap. 5) or configure in-guest security and memory access policies, which allow to protect sensitive data in kernel and user space (Chap. 6).

Kernel mode: Our target system architecture reserves the kernel mode for general-purpose OS kernels inside VMs. Even though our work has a strong affinity towards the Linux kernel for its open source character, the general concepts are applicable to any OS. Depending on whether a domain is used for dynamic (malware) analysis or for general OS hardening, the kernel's level of agnosticism varies with regard to the underlying VMM and hardware. In the former use case, the `DomU` runs an unmodified OS kernel; a VMI framework in `Dom0` utilizes the VMM to analyze malware in `DomU`, without revealing any in-guest analysis artifacts (Chap. 5). In the latter use case, `Dom0` runs a modified kernel that actively interacts with the VMM through the hypercall interface to either activate the hardware's capabilities in the VM, or to leverage the VMM to gain the virtualization-assisted memory isolation capabilities (Chap. 5 – 6).

Aligning the contributions: Our contributions equip selected subsystems of the kernel to protect sensitive OS components against attacks. We focus on hardening the memory and process management system through the virtualization-assisted primitives established in system mode (Chap. 6). Additionally, we pave the way towards enhancing the security of the driver architecture, and obstructing selected attacks against the OS kernel heap management structures, which we highlight for the future work (Sec. 8.2). The extended OS kernel memory management system interfaces the VMM in system mode. We use the system's hypercall interface to propagate access to the strong memory isolation capabilities into the memory management system. Our idea is to repurpose the system's virtualization extensions to lend the kernel memory management the ability to define disjoint and isolated memory domains. We intend to utilize such domains to protect selected sensitive and security-critical data structures against data-oriented attacks (Chap. 6). In this context, we establish the means to thwart attacks against page tables and process credentials of the process management subsystem. Further, our contribution utilizes and extends the Linux namespaces to protect data structures against unauthorized access attempts originated in specified namespaces (Chap. 6). Thus, our contributions also extend the kernel to enhance the isolation capabilities of containers (Sec. 2.1.2).

User mode: The least-privileged abstraction layer of our target system architecture mainly accommodates the user space plumbing layer of the OS. It defines an execution environment for individual or groups of processes in form of containers (Sec. 2.1.2).

Aligning the contributions: The OS kernel extensions pass the virtualization-assisted memory isolation primitives through the system call interface to user space components. This allows processes to protect sensitive data in isolated domains against data-oriented attacks (Chap. 6). Additionally, our target architecture applies system call filtering mechanisms to reduce the number of system calls authorized to processes and containers (Sec. 2.1.2). To accommodate the system call filtering facility, we foresee a static analysis based framework for deriving accurate policies from binaries to restrict a large portion of system calls (Chap. 7).

3.2 Comprehensive Threat Model

To accompany the introduced high-level target system architecture, in this section, we outline the associated comprehensive high-level threat model. Specifically, we distinguish between two distinct roles describing *offensive* and *defensive* tactics of the malicious actor, and the actions she can take in targeting the system. We adapt and extend these roles in the following chapters in order to clarify the threat model and adversarial capabilities of the respective contribution (Chap. 4 – 7).

Offensive attacker: The first role defines the capabilities and intent of an offensive attacker who abuses, e.g., latent memory corruption vulnerabilities to leak or modify sensitive information, or lever out active security mechanisms and take control over the system. In both cases, the attacker aims to establish *read and write* primitives to craft and stitch together gadgets (in user and kernel space) required to conduct various attacks. While the gained primitives in user space allow malicious actors to directly access the victim process' address space, the attacker leverages potentially vulnerable system calls or kernel drivers to indirectly leak or corrupt the kernel memory. The proposed target system architecture assembles a set of virtualization-assisted OS security mechanisms (`Dom0` in Fig. 3.1) to reduce the system's attack surface or even prevent such operations.

Defensive attacker: The second role describes a defensive attacker who utilizes sophisticated deobfuscation, anti-debugging, and anti-virtualization techniques focusing on revealing in-guest analysis environments and artifacts. In other words, the attacker lends malware a split personality to evade the detection and, in particular, analysis of the employed techniques. That is, the attacker is indifferent to virtual environments, yet, she will abort the operation upon disclosing any evidence of an analysis framework. The system architecture reserves and dedicates an unprivileged domain (`DomU` in Fig. 3.1) to inspect and analyze the malicious behavior of such actors in a stealthy way, without noticeably intervening in the malicious actions, and without revealing the analysis framework.

Both roles assume a strong attacker with `root` privileges, yet, without physical access to the target system. Also, in both cases, we assume the adversary operates in a top-down manner by paving the way from user-controlled entities, e.g., from a compromised or implanted malicious user-space process or device driver into the heart of the OS kernel. That is, at the point of the attack, the adversary does not rely on malicious, or otherwise compromised components residing in the same or higher privilege levels (i.e., modes) than the VMM itself. Further, we disregard attacks that require physical access to the system's hardware. In a similar way, we deem the class of attacks out of scope, which abuse Direct Memory Access (DMA) channels (that originate from the outside of the VM) to gain direct access to the guest's physical memory.

3.3 Summary

In this chapter we have defined an abstract, hardware-independent representation of the target system architecture and the associated high-level threat model. To ease the structure of both main research directions pursuit by this work (Sec. 1.2), we logically separate their focus and responsibilities between two VMs of the outlined system architecture. Therefore, we dedicate one VM to organize the involved OS components and virtualization-assisted primitives that are necessary to prevent sophisticated attacks against the OS's kernel and user space processes. At the same time, we dedicate another VM to establish a sandboxed environment equipped with stealthy monitoring primitives that can assist security experts in analysing the behavior of sophisticated malware. Accordingly, we outline the anticipated threats of the offensive and defensive roles or strategies an attacker can pursue. We identify the attack vectors of the offensive and the tendency towards identifying analysis frameworks of the defensive schemes, and finally associate the system's virtualization-assisted components that aim to counter both strategies. We expand and adapt the high-level threat model to the introduced primitives in the following chapters. Overall, the choice to clearly group and position the objectives of both research directions, allows us to create two isolated, yet comprehensive perspectives on our work, in which we leverage virtualization either for stealthy dynamic binary analysis or OS security as described in the remainder of this work.

Chapter 4

On-demand Deployment of Virtualization-assisted Frameworks

Reality is merely an illusion, albeit a very persistent one.
— ALBERT EINSTEIN

Modern malware can acquire and execute with the same privileges as the sensitive parts of the OS. Once installed, it can hide from the OS and its security frameworks. The growing complexity of modern malware drives security experts to increasingly leverage virtualization techniques, which provide access to the complete and untainted view over the VM's state [CN01, GR03, DRSL08, PSE11, Lit08, GDXJ11, LMP+14]. Specifically, they relocate security primitives into a high-privileged VMM [CN01]. This strategy allows security experts in academia and industry to enhance security frameworks with virtualization-assisted memory isolation, inspection, and interposition capabilities (Sec. 2.3). For example, on the industrial side, Microsoft has identified and demonstrated the effectiveness of virtualization-assisted security primitives. In fact, Microsoft integrates the Hyper-V VMM into the OS to enhance its security since Microsoft Windows Server 2008 [Mic17a, YIRS17]. Yet, Microsoft Windows is one of the only few commercial OSes which follow this path.

To benefit from virtualization technology, a VMM has to be explicitly set up underneath the OS. This, however, constraints a wide adoption of virtualization-assisted security frameworks. We believe that as long as OSes are not shipped with an integrated, and possibly minimalistic VMM, which offers an interface to the hardware-assisted virtualization extensions,[1] the popularity of virtualization-assisted security mechanisms in non-cloud environments will remain in its limits. Consequently, our first contribution explores a new deployment strategy for thin VMMs, which, we believe, can influence the design and enhance the capabilities of OS security subsystems.

[1] Note that the Linux kernel implements Kernel-based Virtual Machine (KVM) [KKL+07, Lin20b], a kernel-based VMM. Yet, KVM transforms the Linux kernel into a (Type II) VMM, instead of equipping it with security services.

In this chapter, we elaborate a new deployment strategy and architecture of a thin VMM, which we coin WhiteRabbit. We equip WhiteRabbit with on-the-fly virtualization techniques, which allow the VMM to be deployed on-demand, e.g., on devices in managed corporate or IoT infrastructures. More precisely, we can deploy WhiteRabbit by loading a kernel module, which moves a live OS into a dynamically initialized virtual environment. This deployment strategy can also be implemented as an integral component of the OS kernel. Once deployed, WhiteRabbit transparently unfolds a minimalistic and extensible microkernel-based architecture underneath a running OS, without leaving any traces inside the virtualized guest environment. To demonstrate the flexible deployment strategy for arbitrary virtualization-assisted frameworks, in this chapter we exemplify one specific use case, in which we utilize WhiteRabbit to expose VMI services, which can be transparently deployed for the purpose of forensic analysis. In this context, once deployed WhiteRabbit offers a LibVMI interface that enables it to be engaged by VMI applications. Contrary to existing VMI solutions, our system does not require the target OS to be set up for VMI in advance. Instead, we deploy WhiteRabbit spontaneously on general-purpose systems. As a result, WhiteRabbit transforms the target system into a monitored environment that can be controlled by custom or existing VMI monitors.

Our conceptional design and prototype can take control over the virtualization extensions underneath a running Linux system on Intel as well as ARM (Sec. 2.2). Even though WhiteRabbit's on-the-fly virtualization capability and limited virtualization overhead constitute an effective solution for malware detection and analysis, we consider the deployment strategy of WhiteRabbit as a generic means to set up the necessary environment for arbitrary virtualization-assisted security measures. In other words, we regard WhiteRabbit as a generic vehicle for deploying various virtualization-assisted OS security mechanisms. In fact, we can apply the techniques described in this section to transparently set up other variants of VMI-based binary analysis frameworks (Chap. 5) underneath an executing OS, or dynamically equip the OS itself with additional security primitives (Chap. 6), which we introduce in the following chapters.

Note: We have published parts of this chapter in [PKZ18]. Even though the publication builds upon the ground work defined in the author's Master's thesis [Pro16], the author has reimplemented many parts of WhiteRabbit for x86 and introduced a new prototype for ARM. All measurements stem from the revised and published work, and not the thesis.

4.1 Threat Model

Even though we can apply the on-the-fly deployment strategy of WhiteRabbit to deploy various virtualization-assisted frameworks, throughout this chapter, we focus on the VMI capabilities of our system. Hence our threat model considers and extends the *defensive* attacker strategy, which we have outlined in Sec. 3.2. Overall, the main objective of the adversary is to avoid revealing the internals of her attack. Thus, before mounting her attack against the system, she prefers to ensure the absence of a VMI-based monitor.

We assume an adversary with root privileges, who can fully control the guest OS and all security-relevant parts of the kernel. We disregard how the adversary gains root privileges in the first place to emphasize the stealthy nature of WhiteRabbit. The attacker can inspect the OS for agents in form of processes or kernel modules, which would reveal a security framework. Further, the adversary can inspect the kernel's data structures for potential analysis artifacts and inconsistencies, which would indicate an external monitor. Generally, she can employ anti-debugging and anti-virtualization techniques that reveal the presence of a virtualization-based analysis framework. At the same time, the attacker cannot perform DKSM attacks by exploiting the semantic gap to evade VMI [BJW+10]. The attacker is not concerned about generally virtualized systems, yet, she will abort her attack as soon as she reveals an analysis framework. Even though she can carve the guest's memory, she cannot use DMA or operate with higher privileges than WhiteRabbit. Also, she disregards side channel attacks against VMMs, especially those that base upon a flawed CPU microarchitecture to reveal the VMM's memory from unprivileged execution environments [WVBM+18, Cor20b, SLM+19]. Further, the attacker has access to the system's registers and can search the file system for indications of an analysis framework.

Although WhiteRabbit provides a stealthy environment, VMI applications that are built upon it may employ services detectable by the adversary. For instance, WhiteRabbit does not provide any means to cloak in-guest instrumentation. For this, WhiteRabbit could be extended to support multiple guest-physical memory views to satisfy integrity checks as presented by complementary work based on the Xen `altp2m` subsystem [dml15, LMP+14, SMJ+17, PLM+18, PMG+20] or similar techniques [DZX13], which manipulate or switch among different SLAT tables to coordinate the guest's view on the guest-physical memory. We explore stealthy instrumentations through alternate SLAT configurations in Chap. 5.

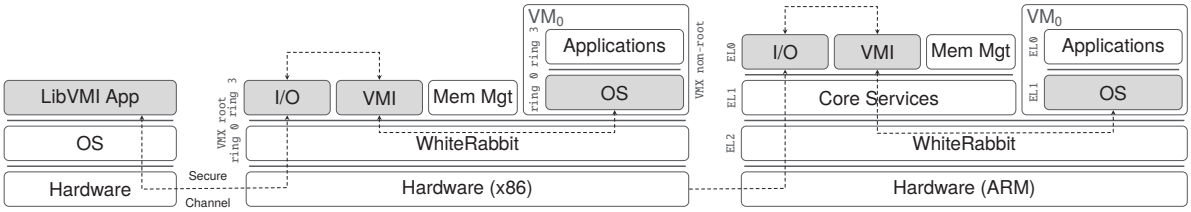


Figure 4.1: WhiteRabbit allows a remote host (left) to analyze the on-the-fly virtualized system in the middle (x86) and right (ARM). Shaded components are involved in VMI.

4.2 The WhiteRabbit VMM

WhiteRabbit comprises a self-sufficient and thin microkernel-based VMM, which we have designed to dynamically virtualize live OSes. As we discuss throughout this work, the concepts behind on-the-fly virtualization allow us to equip various security frameworks with flexible on-the-fly virtualization capabilities. Before diving into details, in the following, we provide a bird's eye perspective on WhiteRabbit and exemplify one use case, in which we consolidate WhiteRabbit's on-demand deployment strategy with VMI.

Generally, to shift a running OS into a virtual environment, our system leverages the virtualization extensions of the Intel and ARM architecture in a special way. As such, we draw upon characteristics of both architectures throughout the remainder of this chapter. Even though we could have adapted traditional VMMs, such as the Xen Project hypervisor [BDF+03, Lin20a] and Linux KVM [KKL+07, Lin20b], to facilitate on-the-fly virtualization of OSes, we have opted for a custom, minimalistic VMM implementation. The main reason for this design decision is that traditional VMM implementations specialize in a different scope; they entail a considerable amount of potentially error-prone functionality that is not necessary and would remain unused when applied for our purpose.

Fig. 4.1 illustrates a simplified architecture of WhiteRabbit on both x86 and ARM. The presented WhiteRabbit software architecture comprises custom subsystems and supplies VMI capabilities to remote parties. To tackle potential exposures, WhiteRabbit hides in memory from the virtualized OS. Specifically, since WhiteRabbit controls the VM's view on the guest-physical memory (Sec. 2.2.3), it excludes its code and data region mappings in the SLAT tables. Further, WhiteRabbit implements and exposes a LibVMI-compatible interface to remote VMI tools. In this way, WhiteRabbit allows remote VMI tools to introspect the dynamically virtualized OS, which has not been explicitly set up for VMI. The microkernel character of WhiteRabbit reduces the size and complexity of the VMM. In fact, we implement only essential VM-maintenance functionality inside the high-privileged protection `ring 0` on Intel and across `EL1` and `EL2` on ARM (Sec. 2.2.1). We place additional components required for memory management, remote communication, and VMI into the user space in `ring 3` on Intel and `EL0` on ARM. This architectural choice allows to isolate WhiteRabbit's user space components from the guest and, at the same time, hardens the system; crashes of user space components do not affect the entire system.

As part of its initialization, WhiteRabbit cuts off any dependencies to the virtualized OS. This includes the OS's services as well as any resource management structures that are bound to the hardware (e.g., interrupt management and paging). Therefore, WhiteRabbit is not bound to the OS's memory or other resource management systems which could be otherwise observed and manipulated by adversaries. Consequently, since WhiteRabbit uses only custom subsystems, which do not require any services of the guest OS, it can generally host arbitrary OSes. For instance, if we provided an OS-independent deployment strategy for WhiteRabbit, our system could virtualize the respective OS on-demand.

The memory management system of WhiteRabbit is a relevant example for the custom subsystems. To satisfy memory allocation requests, this subsystem has to allocate and extract sufficient memory from the guest in the first place. To claim the memory, WhiteRabbit utilizes the (zoned) buddy allocator of Linux during its initialization. The buddy allocator manages the system's physical memory and does not store any relevant metadata that could potentially reveal the VMM; contrary to the Linux SLUB allocator, memory allocated through the buddy allocator does not indicate the purpose of the allocation itself. Instead, the system marks the memory as reserved, and hence not available. Alternatively, to increase its stealthiness during initialization, WhiteRabbit could scan and directly adjust the system's page tables, and manually reserve the system's memory management structures to avoid inconsistencies. Either way, since the VMM leverages SLAT tables to hide its components, the allocated memory regions become invisible to the guest OS after WhiteRabbit completes its initialization. In other words, once set up, WhiteRabbit can intercept the guest's access attempts to the respective memory and, e.g., emulate or redirect the accesses to a non-suspicious memory region [LMP+14].

To remotely access the VMI functionality, the conceptual design of WhiteRabbit dedicates I/O drivers that establish and maintain a secure communication channel to the remote parties. The communication channel should be either entirely cut off or multiplexed with the guest's drivers. To isolate the communication channel the system could employ, e.g., unused I/O devices or hardware extensions that allow to multiplex I/O resources (e.g., through Intel VT-d or VT-c, and ARM System MMU (SMMU)).

4.2.1 On-the-Fly Virtualization

To dynamically virtualize a running OS, we have to deploy WhiteRabbit on the target system in the first place. In this context, we distinguish between *OS-dependent* and *OS-independent* deployment strategies. Both can be performed locally or remotely. The OS-dependent strategy requires a kernel module to set up WhiteRabbit underneath the target OS. The kernel module can (i) either implement the VMM functionality or (ii) merely act as a means for transportation. The former approach implements the WhiteRabbit functionality as part of the kernel module. This strategy must be regarded critically as it allows WhiteRabbit to use the services of the target OS: employed services might reveal the presence of WhiteRabbit or provide false information, which is controlled by malware. On the other hand, this strategy is a valid alternative for virtualization-assisted in-guest

security frameworks that merely require WhiteRabbit to set up and establish an interface to the system's virtualization extensions. We discuss this application scenario in more detail in Chap. 6. The latter approach uses the kernel module as a loader to deploy WhiteRabbit in form of an OS-independent binary into memory. In this scenario, even though the loader requires OS services, WhiteRabbit can remain completely OS-agnostic.

The OS-independent strategy uses a DMA channel to inject WhiteRabbit into the system's memory. By taking the role of the bus master, hardware devices can initiate the communication and hence arbitrary access (assuming deactivated IOMMU) another node's memory. Thus, we could use DMA-capable interfaces (e.g., FireWire or Thunderbolt) to transparently load WhiteRabbit into the system's memory. Yet, after injecting the code, the system requires additional means to execute it. One idea is to use Intel Active Management Technology (AMT) to launch WhiteRabbit from a remote system.

Once deployed on the target system, WhiteRabbit takes on several tasks to virtualize and satisfy the requests of a running OS. These tasks are best described according to the taxonomy of Popek and Goldberg [PG74] (Sec. 2.1.3.2). Their taxonomy describes a VMM as a modular *control program*, whose modules belong to three groups comprising an *allocator*, a *dispatcher*, and an *interpreter*. Accordingly, we distribute the tasks of WhiteRabbit among these groups and describe them in the following. Note that, for simplicity, we assume that WhiteRabbit has been deployed on the target system in form of a kernel module.

4.2.1.1 The Allocator Module

The allocator of WhiteRabbit moves an executing OS into a less-privileged, virtual environment. This module is responsible for setting up the remaining components of WhiteRabbit and configuring the hardware to ensure that the target OS continues with its operation inside a VM, without being aware of any change of its environment. To achieve this, the allocator leverages the system's hardware-assisted virtualization extensions. Specifically, the allocator takes a snapshot of the OS's state and uses the hardware to allocate and configure a virtual environment to reflect the recorded state of the OS. At the same time, the allocator unfolds and registers other modules of WhiteRabbit inside a high-privileged execution environment underneath the VM. In other words, the allocator abuses the system's virtualization extension to shift the OS into a VM, and, at the same time, to provide it with the illusion of having unrestricted access to the system's resources. Overall, this process is highly hardware-dependent. As such, in the following, we discuss the steps necessary to shift an OS into a VM on Intel and ARM (Sec. 2.2).

The Intel architecture: On Intel, the allocator records the system's state (i.e., before it has virtualized the OS) in the guest-state area of the VMCS (Sec. 2.2.2). This area holds control registers that determine the guest's behavior. Also, the allocator sets up the host's state and registers the entry point of the VMM that will be executed at every VM exit in VMX root. Finally, WhiteRabbit grants the VM direct hardware access by passing through devices, without emulating any hardware resources.

The ARM architecture: While the virtualization support of the ARM architecture closely resembles its x86 counterpart, it entails peculiarities, which differentiate the process of on-the-fly virtualization of a live OS. For instance, ARM cannot initialize the exception level `EL2`, which forms a dedicated execution environment for the VMM (Sec. 2.2.2), from a less-privileged exception level: in case the system has not set up the exception vectors in `EL2` at system boot, there is no way to retrospectively place these vectors. Consequently, to allow OSes to access and configure the exception vectors in `EL2`—which are necessary for system virtualization—just before entering `EL1`, the boot loader launches the OS kernel in `EL2`. There, the OS installs a general-purpose hypervisor stub, the *lowvisor*, which initializes the aforementioned exception vectors [DN14]. This lowvisor allows Linux subsystems in `EL1` (e.g., Linux KVM) to retrospectively initialize the exception vectors through a hypercall and take control over `EL2`. Thus, after ensuring that the lowvisor has not been occupied, the allocator of WhiteRabbit uses the same strategy to take control of `EL2`, and hence initializes WhiteRabbit in the high-privileged execution environment underneath the OS.

On both architectures, the allocator configures the system's virtualization-extensions to cause selected guest events of interest to trap into the VMM. Generally, this strategy allows, e.g., VMI frameworks (Chap. 5) to intercept relevant guest events for analysis purposes, or assist in-guest security mechanisms with the capability to provide strong memory isolation capabilities (Chap. 6). Examples for potentially relevant guest events include hardware events, the execution of sensitive instructions (which would otherwise not unconditionally trap into the VMM), and access to the sensitive system registers. Besides, the allocator sets up subsystems (e.g., memory management and device drivers) to manage the system's hardware resources and thus to allow WhiteRabbit to become independent from the virtualized OS. Once the allocator has set up the VM and the individual subsystems of WhiteRabbit, it uses the system's SLAT tables to unmap its code and data regions and hence hide from the guest in memory.

4.2.1.2 The Dispatcher Module

The dispatcher of WhiteRabbit is the entry point of the VMM. Every time the VM exits and transfers the execution to the VMM, the system triggers the dispatcher of WhiteRabbit. This applies to explicitly and implicitly generated VM exits. Explicit VM exits result from explicitly causing the VM to exit, e.g., through hypercalls. Implicit VM exits include all guest events that automatically trap into the VMM. In both cases, the dispatcher analyzes the VM exit reason, based on which it decides which operation to perform next. It is the interpreter (described in the following section) that is responsible for performing tasks on behalf of the dispatcher.

4.2.1.3 The Interpreter Module

The interpreter simulates guest instructions and hardware events that trap into the VMM. Hardware-assisted virtualization extensions define a class of *unconditionally* and *condition-*

ally trapped instructions. The former class comprises privileged instructions that always trigger VM exits when executed inside the VM. For instance, the `CPUID` instruction on Intel unconditionally traps into the VMM. The latter class of instructions trigger VM exits only if the allocator has configured them to trap. The same applies to hardware events. In all these cases, the interpreter provides the necessary functionality to handle the trapped event and appropriately update the guest's state. For this, the interpreter leverages the memory and device management services of WhiteRabbit that are set up by the allocator. In regard to the VMI capabilities of WhiteRabbit, we can utilize the interpreter's ability to manipulate the guest's state to enforce the requests of the VMI subsystem.

4.2.2 Bridging the Semantic Gap

The on-demand deployment strategy of WhiteRabbit allows to dynamically deploy versatile virtualization-assisted frameworks. In other words, we do not limit WhiteRabbit to providing only one specific service. Instead, we dedicate WhiteRabbit to be a fundamental building block that can assist various security mechanisms. Depending on their purpose and policy, these mechanisms can interface WhiteRabbit from inside (Chap. 6) or from the outside (Chap. 5) of the VM. Both concepts have different requirements with regard to accessing the guest's state. For instance, in-guest security mechanisms do not necessarily need to rely on the VMM to reconstruct the guest's state. On the other hand, this assumption changes, when we put WhiteRabbit into the context of VMI-based frameworks; VMI frameworks intend to analyze the guest from an external position and hence have to map the guest's binary state to a semantically-enriched representation (Sec. 2.3). In such cases, we have to ensure that WhiteRabbit implements the necessary primitives, which can assist local and remote VMI frameworks in bridging the semantic gap (Sec. 2.3.2).

We demonstrate the effectiveness and usefulness of WhiteRabbit by demonstrating its ability to accommodate VMI frameworks. Specifically, in the following use cases, we outline example implementations of the (i) *in-band* and (ii) *out-of-band* delivery as well as (iii) *derivative* view generation patterns [PSE09, Pfo13] (Sec. 2.3.2.1). We have implemented these patterns in our prototype to bridge the semantic gap. Note that the following patterns are use cases, which we have selected to demonstrate the applicability of WhiteRabbit. We highlight that WhiteRabbit is a generic vehicle for virtualization-assisted security mechanisms, which we demonstrate in the context of VMI frameworks. That said, VMI frameworks which build upon WhiteRabbit, can implement custom policies and use WhiteRabbit to enforce them by applying the presented, or other implementations of the view generation patterns, which are not limited to the following use cases.

4.2.2.1 In-band Delivery Pattern

This pattern involves the guest OS to collect semantic information (Sec. 2.3.2.1). We implement one specific use case, which adopts the concepts of the X-TIER framework [VKSE13, Vog15]. Specifically, WhiteRabbit allows remote VMI tools to inject kernel modules into

the guest. Before WhiteRabbit can inject the module, its VMI component (Fig. 4.1) has to process the module to establish a generic and OS-agnostic module representation (as we do not claim any novelty on the OS-agnostic module representation, we refer the reader to the X-Format of the X-TIER framework [VKSE13, Vog15]; our prototype uses a format that closely resembles the X-Format). To simplify the implementation of the VMI component, these steps could be prepared in advance by the remote host.

Generally, to inject kernel modules into a VM, WhiteRabbit has to undergo multiple steps. First, the VMI component has to allocate and map additional memory into the VM to position the module's sections. This poses a challenge, since the memory management system of the dynamically virtualized OS has tracked the available physical memory. In contrast to WhiteRabbit, X-TIER operates as part of Linux KVM, a Type II system VM (Sec. 2.1.3). As such, it can request memory from the underlying OS to accommodate this step. On the other hand, WhiteRabbit cannot simply add physical memory to the VM. Instead of resorting to the services of an underlying OS, it obtains the memory from the local memory pool, which it has extracted from the guest during its deployment (Sec. 4.2).

In the next step, the VMI component of WhiteRabbit equips the module by additional functionality that is required to communicate with WhiteRabbit from within the guest. The added functionality comprises wrappers responsible for relaying calls to exported kernel functions and announcing the end of the module's execution through hypercalls.

In the final step, WhiteRabbit temporarily uncovers the module in the guest's physical memory by means of the SLAT tables; injects the module into the address space of the interrupted guest process by adjusting its page tables; and redirects the guest's instruction and stack pointer to transfer the guest's control-flow to the module. As such, upon resuming the VM, the guest will execute the injected module. To prevent the module from being interrupted, and thus potentially revealed, WhiteRabbit must take additional precautions. In the simplest case, it could deactivate the guest's timer interrupt and intercept external interrupts as long as the module executes. An alternative solution could configure a set of SLAT tables to establish different views on the guest's physical memory and effectively hide the module from the guest. We provide an example of a similar configuration in Chap. 5.

4.2.2.2 Out-of-band Delivery Pattern

WhiteRabbit implements an interface for LibVMI [Pay12, Lib20]; a C-library that implements an API to dynamically extract and control the VM's state. To bridge the semantic gap, LibVMI uses out-of-band kernel symbol information that is delivered ahead-of-time. This library further complements the out-of-band delivered information with knowledge of the VM's system architecture. Similarly, LibVMI offers an API for the memory forensic analysis framework, Volatility [Vol20, LCLW14]. In this way, WhiteRabbit offers an effective interface for custom and prevalent LibVMI and Volatility based forensics tools.

4.2.2.3 Derivative Pattern

VMI frameworks that leverage the derivative view generation pattern [Lit08, DRSL08, PSE11], engage in-depth knowledge of the guest's hardware architecture to derive the guest OS's semantic information. Derivative view generation benefits from the fact that critical static in-guest data structures are rooted in hardware [PSE11, Pfo13]. For instance, one can build a chain of references between the system call dispatcher and an immutable hardware anchor (Sec. 2.3.2.1); on Intel, the `IDTR` or the MSRs that are related to fast system calls (e.g., `SYSENTER_EIP_MSR`) present immutable hardware anchors, which root the system call dispatcher in hardware. Unfortunately, it is difficult to identify and utilize such hardware anchors. Yet, through LibVMI, WhiteRabbit facilitates remote tools to use their knowledge of the hardware architecture to derive the guest's view. In fact, a derived view can be accommodated by additional properties: namely *evasion-evidence* and *evasion-resistance* [PSE11, Pfo13]. According to Pfoh et al., critical guest data structures are *evasion-evident* if they are rooted in hardware. This property allows WhiteRabbit to observe changes to such data structures. By additionally protecting all elements along the chain from the hardware anchor to the data structure, the data structure becomes *evasion-resistant*: any modification along this chain can be detected by matching the integrity of the system's configuration with a known value [PSE11, Pfo13].

4.2.3 Hiding Techniques

The inspection capability of VMMs lends VMI frameworks an omniscient character (Sec. 2.3). This character ensures that malicious behavior inside VMs cannot easily mislead a VMI-based analysis. As such, VMI has become in particular attractive in analyzing sophisticated malware. In an attempt to counter the analysis, malware applies anti-debugging and anti-virtualization techniques to scan its execution environment for artifacts that would reveal a monitor [CAM+08, BCK+10, SAM14]. As soon as the malware believes it is being monitored, it exhibits a different personality and changes its behavior. Thus, WhiteRabbit must apply cloaking techniques to avoid exposing itself and the associated VMI frameworks. Given that perfect *VM transparency* (the ability of being indistinguishable with real hardware) is not feasible and impractical for modern defense mechanisms [GAWF07], WhiteRabbit does not intend to eliminate all side effects that are related to virtualization (Sec. 2.3.1). A virtualized system does not always indicate a monitored environment. In fact, the modern trend towards cloud computing makes this assumption obsolete. Consequently, WhiteRabbit mainly focuses on cloaking its presence in the guest's memory, without explicitly hiding any general virtualization artifacts. Specifically, WhiteRabbit sidesteps common anti-virtualization techniques, which focus on the *execution environment* and *hardware* category of the anti-virtualization taxonomy introduced by Chen et al. [CAM+08]. We associate WhiteRabbit's memory footprint with the *execution environment* and traces of the kernel module with the *hardware* category. To sidestep both, WhiteRabbit applies the SLAT mechanism.

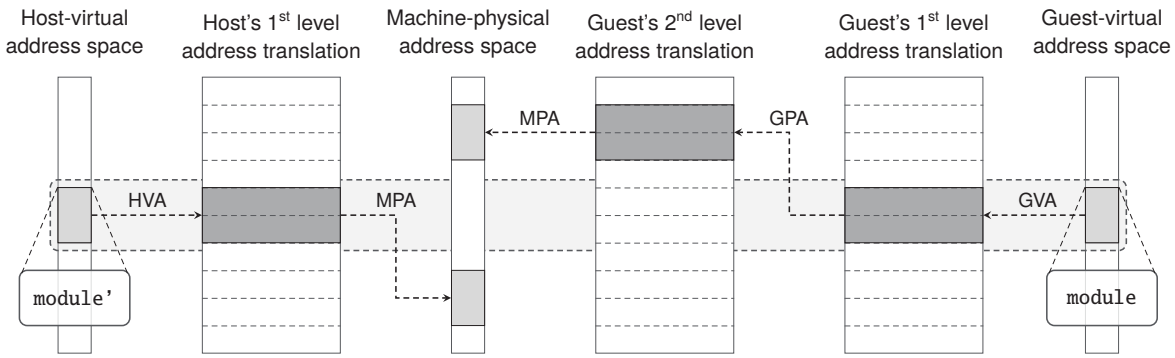


Figure 4.2: WhiteRabbit relocates itself and uses the SLAT mechanism to hide in memory.

4.2.3.1 Execution Environment Artifacts

To expose an analysis framework, a malicious guest can carve the guest-physical memory, e.g., for signatures that could indicate the presence of an in-guest agent of the VMI framework. To prevent an exposure in memory, we utilize the system's SLAT mechanism. Specifically, we unmap the guest-physical memory of WhiteRabbit in the SLAT tables such that our framework becomes invisible to the guest. This strategy allows WhiteRabbit to intercept and satisfy accesses to this memory, by redirecting them to a safe memory region.

At the same time, if we deployed WhiteRabbit in form of a kernel module, this strategy would entail the following challenge. As soon as the kernel module takes control over the system's virtualization extensions to accommodate WhiteRabbit, it will resume the OS. At this point, the OS—now running inside the VM—needs to continue the original control-flow of the kernel module to complete the initial request. In this way, WhiteRabbit can safely resume the OS without having to adjust the kernel's stack or any kernel data structures from within the VMM. Unfortunately, since WhiteRabbit unmaps itself from the guest-physical memory, resuming the guest inside the kernel module would violate the guest's memory permissions in the SLAT table and hence trap into WhiteRabbit.

For clarification, let us depict a scenario, in which the kernel module initiates the process of moving the OS into a VM as part of its initialization function. As soon as the module completes virtualizing the OS (i.e., as part of the VMM), it will need to return to its initialization function (i.e., within the VM) to complete the kernel module's initialization request. In other words, the OS's first instruction within the VM is, at the same time, one of the last instructions of the kernel module that has virtualized the OS. The module's initialization function (inside the VM) needs to return to the Linux kernel to complete the module's initialization. However, at this point, the kernel initializes remaining entries of the `struct module`, which, in turn, resides in the module's memory protected by SLAT. In addition, we have to allow the kernel to release the memory holding the kernel module. In both cases, the kernel needs to access the memory, which has been previously made invisible, or rather inaccessible to the guest OS. In the following section, we discuss the necessary steps required to overcome this challenge.

4.2.3.2 Hardware and Device Driver Artifacts

The hardware anti-virtualization category inspects, among others, the system's device drivers in order to to reveal virtualization-based analysis frameworks. Thus, it is the task of WhiteRabbit to eliminate any hardware, or rather device driver related artifacts inside of the VM. That is, assuming WhiteRabbit was deployed in form of a kernel module, it would need to conceal not only its code and data sections, but also any in-guest data structures that could indicate the presence of the kernel module.

As we have discussed in the above scenario, leveraging the SLAT mechanism alone to cloak the presence of WhiteRabbit leads to an issue during the kernel module's initialization stage. To overcome this challenge, we have to leverage the guest kernel itself to free the memory and data structures that are linked to WhiteRabbit. At the same time, we have to provide the means to facilitate WhiteRabbit's operation under cover of the SLAT mechanism. To achieve this, WhiteRabbit (i) relocates itself to another location in memory and (ii) instructs the guest kernel to release the originally loaded kernel module inside the VM, without affecting its relocated copy.

Fig. 4.2 illustrates the relocation process of the original WhiteRabbit kernel module (`module`) to another location in the machine's physical memory (`module'`). To avoid having to explicitly relocate the virtual addresses inside the code section of `module'`, we map the relocated `module'` to the same virtual address region in the VMM's address space, as the one occupied by the original `module`. To achieve this, we use the following setup. The guest's page tables and the complementary SLAT tables map the guest-virtual addresses to the machine-physical addresses of the original `module`. At the same time, the VMM's page tables map the original virtual addresses to the relocated machine-physical addresses of `module'`. This way, the host can use the original virtual addresses to address `module'` (the VMM's virtual addresses correspond to guest-virtual addresses). This setup allows us to unmap `module'` in the guest-physical memory and hence hide its presence in memory.

To conclude the hiding process, once WhiteRabbit (`module'`) launches the OS kernel, it will resume the initialization function of the original `module` inside the VM. Thus, by returning a negative value at the end of its initialization routine, the `module` instructs the kernel to release its memory and all in-guest data structures that are associated with WhiteRabbit. Alternatively, the module can configure a work-queue that initiates a clean module destruction without making the kernel suspicious. In a final step, WhiteRabbit zeroes out the contents of the released in-guest data structures and memory pages, which used to represent the original kernel module, to ensure that WhiteRabbit cannot be retrospectively reconstructed.

4.3 Evaluation

We evaluated WhiteRabbit by first assessing its practicality in regard to performance. In this regard, we have investigated the induced virtualization overhead and compared it with the popular Xen Project hypervisor [BDF+03, Lin20a] and Linux KVM [KKL+07, Lin20b]. Second, we have analyzed WhiteRabbit's effectiveness. In this part of the evaluation we make clear that common anti-debugging techniques are ineffective against VMI-based analysis frameworks that build upon WhiteRabbit. We extend our discussion by assessing the impact of different anti-virtualization techniques on WhiteRabbit.

4.3.1 System Setup

Our system setup comprises a Linux kernel v4.13. The host is equipped with a 4-core (8 threads) 3.4 GHz Intel Skylake Core i7-6700 CPU, and 16GB of RAM. We have limited the CPU to only one active core to accommodate the implementation-based limitation of our prototype. In fact, to simulate a comparable load among different VMM implementations, we have granted only one core to all VMMs, and pinned the VMM and guest to the same physical core. Finally, we configured the *performance* CPU frequency scaling governor of Linux (and the respective VMM) to avoid performance drops, e.g., due to power consumption oriented configurations.

4.3.2 Performance

It is crucial that the VMM and VMI tools affect the system's performance as little as possible. Since we intend to use WhiteRabbit as a vehicle for generic VMI tools, our performance evaluation focuses on the *virtualization overhead*. That is, in the following evaluation, we do not consider VMI tools that were built upon WhiteRabbit. As the performance highly depends on the respective purpose, we deem the overhead of VMI tools out of scope.

To estimate the virtualization overhead of WhiteRabbit, we carried out three rounds of experiments. All reported results correspond to a vanilla Linux vs. a Linux inside a VM, hosted by three VMM implementations: namely Linux KVM, Xen (with Linux inside the unprivileged `DomU`), and WhiteRabbit. The results are mean values over three runs.

First, we compared the virtualization overhead of WhiteRabbit with Xen v4.11 and Linux KVM. Interestingly, our initial results have shown that Xen outperformed the bare metal Linux, with active Intel Turbo Boost technology. As such, we deactivated Turbo Boost to avoid different microcode decisions in regard to performance states. We have used a set of CPU- and memory-intensive macro (Phoronix v7.6.0 and SPEC CPU2017) and micro (LMbench v3.0) benchmarks to stress different system components and hence to determine the virtualization overhead of Linux KVM, Xen, and WhiteRabbit.

Table 4.1: Virtualization overhead (OHD) of WhiteRabbit, Xen, and Linux KVM, measured using Phoronix v7.6.0 on x86.

Benchmark (*unit*)	w/o	KVM	(OHD)	Xen	(OHD)	WhiteRabbit	(OHD)
Blake2 (*Cycles/Byte*)	5.94	5.94	(0.00%)	5.94	(0.00%)	5.94	(0.00%)
C-Ray (*s*)	107.08	108.56	(1.38%)	107.67	(0.55%)	107.09	(0.00%)
Gzip Compression (*s*)	11.50	12.06	(4.86%)	11.98	(4.17%)	11.74	(2.08%)
John-the-Ripper DES (*Real C/s*)	5,419,000	5,340,000	(1.45%)	5,394,000	(0.46%)	5,417,667	(0.02%)
John-the-Ripper MD5 (*Real C/s*)	16,844	16,583	(1.54%)	16,748	(0.56%)	16,822	(0.13%)
N-queens (*s*)	216.70	220.94	(1.95%)	217.77	(0.49%)	216.80	(0.04%)
OpenSSL (*Signs/s*)	145	141.83	(2.18%)	142.53	(1.70%)	144.70	(0.20%)
7-Zip Compression (*MIPS*)	4,603	3,736	(18.83%)	3,988	(13.36%)	4,443	(3.47%)
RAMspeed Integer (*MB/s*)	17,370.73	16,630.25	(4.26%)	16,942.71	(2.46%)	17,016.09	(2.04%)
RAMspeed Floating Point (*MB/s*)	17,734.84	16,744.56	(5.58%)	16,875.53	(4.84%)	16,861.26	(4.92%)

Table 4.2: SPEC CPU2017, in *sec*.

Benchmark	w/a	WhiteRabbit	OHD
`600.perlbench_s`	282	286	(1.41%)
`602.gcc_s`	409	419	(2.44%)
`605.mcf_s`	624	641	(2.72%)
`620.omnetpp_s`	382	406	(6.28%)
`623.xalancbmk_s`	283	294	(3.88%)
`625.x264_s`	378	378	(0.00%)
`631.deepsjeng_s`	357	363	(1.68%)
`641.leela_s`	460	460	(0.00%)
`648.exchange2_s`	264	265	(0.37%)
`657.xz_s`	2220	2379	(7.16%)

Table 4.3: LMbench v3.0, in μ*sec*.

Benchmark	w/a	WhiteRabbit	OHD
`fork()+execve()`	50.04	58.23	(16.36%)
`fork()+exit()`	47.01	55.58	(18.23%)
`fork()+/bin/sh`	254.36	289.84	(13.94%)
`pipe()`	1.51	1.65	(9.27%)
`read()`	0.09	0.09	(0.00%)
`select()` (500 fds)	2.46	2.52	(2.38%)
`select()` (500 TCP fds)	8.16	8.35	(2.32%)
`write()`	0.05	0.06	(19.99%)
Protection fault	0.30	0.31	(3.33%)
Signal delivery	0.66	0.65	(1.51%)
UNIX socket I/O	1.99	2.07	(4.02%)

Table 4.1 shows the Phoronix results, which we divided into CPU- (upper part) and memory-intensive (lower part) benchmarks. Overall, the results indicate only a minor overhead for all candidates. Yet, WhiteRabbit outperforms Xen and KVM. While KVM produces less than 4.02% CPU and 4.92% memory overhead on average, the virtualization overhead of WhiteRabbit is kept to a minimum at 0.74% for CPU and 3.48% for memory benchmarks on average. According to our measurements, Xen outperforms KVM and approaches WhiteRabbit with an averaged 2.66% CPU and 3.65% memory bandwidth overhead. While we expected the arithmetically-heavy benchmarks, *Gzip* and *7-Zip*, to perform similarly to other CPU-intensive benchmarks, they are outliers for all candidates.

Performance measurements among VMMs can be unreliable as each VMM might emulate and scale the guest's clock source differently. As our prototype does not emulate any clock sources, we can precisely determine the resulting virtualization overhead. Therefore, we ran the *SPECspeed Integer* benchmarks of the *SPEC CPU2017* suite and summarized the results in Table 4.2. Overall, we can see that WhiteRabbit entails only minor performance overhead. Yet, in line with the findings in Table 4.1, the outliers are compression-heavy benchmarks (Gzip, 7-zip, and `657.xz_s`), which were apparent on Linux KVM and Xen.

Finally, we selected a set of *LMbench v3.0* latency-focused micro-benchmarks, to observe the performance overhead on the system software level (Table 4.3). The overall picture suggests that the design of WhiteRabbit is ideally suited as a basis for virtualization-assisted security tools, which do not require the services of a fully-fledged VMM.

4.3.3 Effectiveness

We have virtualized a Linux and were able to single-step and extract analysis-sensitive processes. Therefore, we employed state-of-the-art anti-debugging and anti-virtualization techniques that impede dynamic analysis and stop execution as soon as they believe they reside in a sandbox. The following summarizes these techniques and shows that they are rendered ineffective against WhiteRabbit.

4.3.3.1 Anti-Debugging

Linux bares an API via the `ptrace` system call to allow debugging of user space processes. This API utilizes hardware-based memory watchpoints and single-stepping capabilities, as well as the ability to access foreign address spaces. However, `ptrace` entails that any tracee can be traced by exactly one process. This property is abused for anti-debugging: a hostile machine code can use `ptrace` to trace itself. Consequently, if `ptrace` fails, the caller becomes aware of a tracing application; if it succeeds, no other tracer will be able to attach herself to this process. While this situation can be side-stepped by intercepting calls to `ptrace` and adjusting the return values, the idea can be extended to multiple malicious processes tracing each other to completely hinder debugging.

Besides, debuggers (e.g., *gdb* and *lldb*) leave environment artifacts that can reveal debuggers. These artifacts include (i) address space layout randomization allocating the `text`, `data`, and virtual Dynamic Shared Object (`vDSO`) pages at unusual addresses, (ii) environment variables, (iii) the parent process' name containing the debugger's name, and (iv) software breakpoints non-transparently placed into the tracee's address space. We have open sourced a debugger detection tool implementing the above.[2]

WhiteRabbit does not make use of any of the above techniques. In fact, WhiteRabbit does not leave in-guest user space artifacts and thus cannot be detected by these and similar anti-debugging mechanisms.

4.3.3.2 Anti-Virtualization

To assess WhiteRabbit's ability to evade anti-virtualization techniques, we have armed the virtualized Linux with custom and publicly available sandbox-detection tools including *paranoid fish*, *al-khaser*, and *virt-what*. These tools apply (i) *static heuristics*, (ii) *low-level system properties*, and (iii) *user behavior artifacts* to disclose sandboxed environments [CAM+08, MANP17]. Even though the following evaluation is by no means complete, we intend

[2] `https://github.com/kirschju/debugmenot`

to clarify the strategies that lend WhiteRabbit the ability to cope with common anti-virtualization techniques. Disregarding timing differences in the virtualized environment, given that WhiteRabbit alone does not leave any artifacts inside the guest, none of these tools were able to identify WhiteRabbit.

Static heuristics: Static heuristics target virtualization artifacts (e.g., drivers, execution environment and hardware configuration, vendor information, as well as memory and file system artifacts) that are specific to virtual environments controlled by well-known VMM implementations. Striking MAC addresses are additional indicators for sandboxed environments. Generally, WhiteRabbit does not change the system configuration that is visible to the guest OS. Similarly, WhiteRabbit does not leave in-guest artifacts (including memory and the file system) that could be captured by static heuristics (Sec. 4.2.3). Consequently, WhiteRabbit renders static heuristics obsolete.

Hardware artifacts: Hardware artifacts comprise timing properties and effects of imperfect instruction and device emulation. Further hardware-based information leakages emerge, among others, from register contents. The analysis framework could employ in-guest debug registers and performance counters for analyzing the guest. For instance, as we discuss in Chap. 5, hardware breakpoint and watchpoint register contents can expose the analysis framework. Further, sophisticated analysis frameworks can make use of the hardware control-flow features, such as Intel's Processor Trace [Int20a], to trace control-flow events. Thus, the contents of such debug registers can indicate an underlying analysis framework. As WhiteRabbit alone acts as a vehicle for VMI (and other virtualization-assisted security) applications, it does not make use of such registers. Thus, the VMM itself cannot be detected through leaking register contents. On the other hand, the reader must consider that careless VMI tools, which build upon WhiteRabbit could implement less-stealthy techniques, and hence expose their presence. To address such situations, WhiteRabbit provides the necessary means to intercept critical events. Thus, VMI tools must handle such events and return inconspicuous register values to cloak the analysis.

Besides, WhiteRabbit permits the guest to directly access the hardware without emulating any hardware devices. Thus, it does not expose itself through such indicators. On the other hand, timing differences can indeed reveal the VMM. In fact, we were able to expose WhiteRabbit by comparing the time of unconditionally trapped instructions with reference values. However, with today's omnipresent virtualization technology, it is insufficient to reveal the virtual environment alone (Sec. 4.4.2).

User behavior artifacts: User behavior artifacts target the system's credibility by observing its state and configuration, including mouse cursor activity or an unusually small size of the hard drive or memory. Sophisticated systems check wear-and-tear relics, e.g., log files, browser history, and network behavior [MANP17]. Such artifacts lose relevance, as WhiteRabbit virtualizes production systems with realistic wear-and-tear relics.

4.4 Discussion

WhiteRabbit is a powerful tool that might turn into a serious threat in hands of adversaries. Therefore, in the following, we discuss countermeasures to defend against its malicious use. Additionally, we turn the attention towards limitations of our WhiteRabbit prototype.

4.4.1 Countermeasures

A proactive approach against adversaries with access to WhiteRabbit suggests to employ a native VMM, such as Xen, that executes directly on bare metal and hence occupies the system's virtualization extensions. This approach can obstruct malicious attempts to subvert the entire system. Since the native VMM would host a set of VMs, the attacker with `root` privileges inside one of the VMs, would not be able place WhiteRabbit underneath the native VMM. Assuming an attacker attempts to initialize WhiteRabbit from a compromised VM, the underlying VMM will be able to intercept and discard any subversion attempts: on Intel, the instructions required to set up the VMX root operation will implicitly trap into the VMM; on ARM, the VMM will deflect any attempts to reconfigure `VBAR_EL2`. In both cases, the VMM would be able to detect the malicious pattern and obstruct the attack. Even if the maliciously utilized WhiteRabbit supported nested virtualization (enabling VMM hierarchies), it would not be able to take exclusive control over the system's virtualization extensions as they would be occupied by the benign VMM. The same applies to hosted VMMs, such as Linux KVM: in this scenario, subversion attempts from a compromised VM would not be able to take over control over an operating VMM. On the other hand, an adversary could subvert the entire system as long as Linux KVM has not taken control over the system's virtualization extensions. This is true for both Intel and ARM.

Given the technological advancements of both Intel and ARM architectures with regard to nested virtualization, the following question arises. Assuming the underlying VMM implementation supported nested virtualization, would it be still possible to subvert the compromised guest and position WhiteRabbit in a nested way in between the VM and the VMM? Although the native VMM would intercept every (nested) virtualization attempt, without additional precautions and guest-behavior analysis, we assume that the VMM would not hinder WhiteRabbit from subverting the guest (much like it would not hinder a second level, i.e., nested, virtualization). In other words, we strongly believe that WhiteRabbit would be able to move the compromised VM into a nested virtual environment, and position itself between the first level VMM and the (nested) VM. In this context, WhiteRabbit would need to configure the system to forward traps of the nested VM to the nested WhiteRabbit—which is the default setting in nested environments. We leave the question of how to defend against such scenarios for future work.

4.4.2 Limitations

Depending on the point of view, we identify a set of limitations that affect different vectors. These vectors target VMI in general as well as the implementation deficits of WhiteRabbit. Given that the dynamic deployment of VMI frameworks through WhiteRabbit presents the main use case of this chapter, we first amplify upon one of the main considerations that affects VMI, namely VM transparency (Sec. 2.3.1).

Malware can evade analysis through anti-virtualization techniques [CAM+08, BCK+10, MANP17]. These consider, among others, side effects of emulated instructions, as certain instructions are not sufficiently documented [Dom17]. To address potential inconsistencies between virtual and physical environments, one can attempt to make the appearance of VMs indistinguishable from real hardware; a security analyst can approximate the behavior of the hardware by gaining the necessary domain knowledge through massive testing [SAM14]. Regardless of how well the virtual environment manages to mirror the appearance of a physical machine, differences in timing behavior will always remain. A system that achieves perfect VM transparency is not feasible and impractical [GAWF07] (Sec. 2.3.1). In other words, adversaries, e.g., with access to external time sources will always be able to detect discrepancies caused by virtualization. Yet, the trend toward system consolidation through virtualization renders the goal of VM transparency obsolete. If a system is virtualized, it does not necessarily mean the malware is subject to analysis. Thus, it is more affordable for attackers to target both physical and virtual environments than exclusively focusing on physical machines. Consequently, any VMI framework that leverages WhiteRabbit should focus on concealing *analysis artifacts*, instead of the artifacts and inconsistencies, which can arise due to virtualization. Since the question of stealth is vital for VMI, in Chap. 5, we introduce new methods that assist VMI tools in cloaking analysis artifacts.

The next limitation of VMI, in particular when combined with WhiteRabbit's deployment strategy, refers to the strong semantic gap [JBZ+14] (Sec. 2.3.2.2). Even though the combination of in-band and out-of-band delivery with derivative patterns establishes a solid ground for analysis (Sec. 2.3.2.1), this combination cannot detect every modification performed by VMI-aware malware. The reason for this is that delivery-based view generation patters strongly rely upon the guest OS or external information delivery sources; at the same time, derivative approaches cannot reconstruct the entire state to fully eliminate the need for a trust anchor [PSE09]. This is because not every data structure can be bound to hardware. Consequently, unannounced structural modifications of these data structures (e.g., through malicious relocation in memory) may remain unnoticed [BJW+10]. The same applies to OS infections that have happened before the analyst was able to deploy her VMI framework via WhiteRabbit. Since WhiteRabbit could have missed the point of infection, it cannot assume the guest OS is trustworthy at the time of its deployment. This implies that VMI tools cannot rely on the guest's integrity as long as every semantically relevant data structure is not bound to hardware or its trustworthiness is not otherwise validated at run-time [JBZ+14]. This is an open challenge that results from the strong semantic gap.

While recent advances in confidential computing have established a basis for separating workloads into components with different trust levels [Int21, Adv20, Arm21], we leave the question of how to establish trust in a potentially malicious environment for future work.

Besides, since WhiteRabbit facilitates dynamic deployment of virtualization-assisted security mechanisms, a deployed VMI tool can miss the actual attack. For instance, attackers can inject one-shot exploits to gather critical information, and unload them before the security analyst has a chance to deploy the VMI tool via WhiteRabbit. The same applies to periodical system checks, which regularly load and unload WhiteRabbit; conducted attacks may slip through periodic system checks and leverage the semantic gap to delude VMI applications [JBZ+14, BJW+10]. These restrictions render WhiteRabbit more suitable for detection and analysis of long-living, persistent malware or for deploying virtualization-assisted security mechanisms that directly support subsystems of the OS. For instance, we could leverage WhiteRabbit to establish an interface between system's virtualization extensions and dedicated subsystems of the guest OS (Chap. 6).

Another limitation affects the cloaking ability of WhiteRabbit. DMA-capable devices have access to the system's physical memory. Through DMA, adversaries can locate WhiteRabbit in memory, despite the SLAT mechanism. To approach this, WhiteRabbit could restrain DMA access by engaging the system's IOMMU (Intel VT-d or ARM SMMU).

4.5 Related Work

Our work on WhiteRabbit combines the deployment strategy of state-of-the-art virtual machine based rootkits (VMBRs) with concepts of VMI. Thus, in the following, we consider previous research, that is related to both concepts.

In-band delivery based frameworks: PI [GDXJ11] is an in-band delivery framework for injecting security applications into a guest VM. Vogl et al. [VKSE13] extend this idea with *X-TIER*, a framework for malware detection and removal. In contrast to PI, which hijacks user space processes, X-TIER injects kernel modules into the guest. Both PI and X-TIER bridge the semantic gap by reconstructing the view on the guest's state from information that is delivered by the injected agents. The VMI component of WhiteRabbit adapts the concepts of X-TIER to inject kernel modules into guest VMs. Yet, WhiteRabbit allows VMI tools to additionally correlate the gathered in-band delivered guest state with information gathered through out-of-band delivery and derivative view generation patterns.

Out-of-band delivery based frameworks: DRAKVUF [LMP+14, PLM+18] is a stealthy, VMI-based, dynamic binary analysis framework. It applies out-of-band delivered symbol information to reconstruct the guest's state and leverages the VMI primitives of LibVMI [Pay12, Lib20] to introspect VMs on x86 and ARM. Since, WhiteRabbit exposes a LibVMI interface to remote VMI frameworks, it could equip DRAKVUF with the ability to dynamically virtualize a system, e.g., for malware analysis. Volatility [Vol20] and Rekall [Rek20] are prominent out-of-band delivery based memory forensics frameworks that facilitate cross-platform memory analysis of VM images and memory dumps.

Derivation based frameworks: *Nitro* [PSE11] introduces a VMI framework that uses its hardware architecture knowledge to derive semantic information about the guest OS. Nitro utilizes virtualization extensions to trace the guest's system calls. *Ether* [DRSL08] manipulates the fast system call dispatcher to redirect the guest's system calls to a fixed, unpaged memory location. In this way, Ether causes system calls to generate page faults, which, in turn, can be intercepted by the VMM. Another derivative view generation approach is taken by Litty et al. [Lit08]. They present *Patagonix*, which is a hash-based memory validation framework on top of Xen. It employs binding semantic knowledge related to the MMU and the paging mechanism to detect malware. Similarly, Kittel et al. [KVL+14] present a Linux kernel validation approach, which considers run-time code patching performed by the kernel. Similarly, we can apply WhiteRabbit's ability to derive the guest's view for kernel validation. Even more, in combination with on-the-fly virtualization, the target OS kernel can be validated periodically through temporal injection and unloading of WhiteRabbit without inducing virtualization overhead between the checks.

VM-based frameworks: *SubVirt* [KCW+06] introduces one of the first VMBRs that can be permanently installed as a VMM underneath existing Linux and Windows OSes. In the meantime, VM-based rootkits have evolved to hardware-assisted VM (HVM) rootkits. Rutkowska introduces *Blue Pill* [Rut06b], an HVM rootkit that is able to transparently move an executing OS instance into a virtual environment controlled by a thin VMM. In parallel to Blue Pill, *Vitriol* [Zov06] present a mostly similar HVM rootkit to subvert Mac OS X on Intel. Later, the *New Blue Pill* [RT07] was presented to also support the Intel VT-x technology. In addition, Cloaker [DCCC08] and CacheKit [ZSS+16] present hypervisor-assisted rootkits for the ARM architecture. Further, Buhren et al. [BVN16] demonstrate attack vectors on ARM that allow to subvert a running Linux on-the-fly.

On-the-fly virtualization based security frameworks: Similar to WhiteRabbit, HyperSleuth [MFPC10] is a small VMM that can virtualize a running Windows XP on-the-fly on Intel. However, contrary to WhiteRabbit, HyperSleuth does not utilize the hardware-assisted SLAT mechanism and thus entails higher software overhead. It also does not hide its in-guest artifacts. This exposes its presence to in-guest malware and thus is not suited, e.g., for analysing sophisticated split-personality malware.

4.6 Summary

In this chapter, we have presented WhiteRabbit, a thin VMM, which is able to virtualize a running Linux OS on-the-fly on Intel and ARM architectures. The flexible deployment strategy of WhiteRabbit allows us to dynamically position virtualization-assisted security frameworks, e.g., on devices in managed corporate or IoT infrastructures. Once deployed, WhiteRabbit unfolds a microkernel-based architecture underneath the OS. We consider WhiteRabbit as a generic vehicle that can equip versatile virtualization-assisted security frameworks with a flexible deployment strategy. To demonstrate the potential behind our framework, in this chapter, we have exemplified a specific use case, in which we have used WhiteRabbit to unify VMI with on-the-fly virtualization—even though we do not limit the presented deployment strategy to VMI. In this context, we have used WhiteRabbit to dynamically place a VMI framework on a general-purpose system that was not specifically set up for VMI in advance. WhiteRabbit has incorporated the system's SLAT mechanism to hide its presence in memory. Further, it has exposed a LibVMI-compatible interface to remote hosts to facilitate VMI-based analysis of the virtualized OS. We have evaluated our prototype on Linux running on-top of Intel. Our results demonstrate that the dynamic virtualization of a running OS is fast and further system virtualization does not present a significant performance overhead.

Having presented the concepts of the generic VMM deployment strategy, in the following chapter, we turn our attention towards our first objective (Q1). Specifically, we investigate new primitives to facilitate, in particular, *stealthy* VMI on ARM devices without the necessary hardware support. We highlight that, even though we leverage the Xen Project hypervisor to enforce the proposed primitives, they could be equally deployed on-the-fly by means of WhiteRabbit.

Chapter 5

Stealthy Monitoring on ARM

Sometimes the problem is to discover what the problem is.
— Gordon Glegg

Virtualization technology can help defenders to gain the upper hand in the arms race against malicious actors. As hypervisors expose a narrow, virtual hardware interface toward guests, they bare a limited attack vector. At the same time, they maintain a complete and untainted view of the guest's state (Sec. 2.3.1). To benefit from this constellation, we can apply virtual machine introspection (VMI) techniques [GR03] (Sec. 2.3). As we have learned in the previous chapters, VMI constitutes techniques that lend security frameworks the ability to observe, analyze, and control the state of VMs. These techniques have evolved over the past two decades and have reached a point, which has become relevant not only to academia but also to the industry [Fir20, VMR20, Bed20]. Recent research on VMI has placed a particular emphasis on the *stealth* property of VMI frameworks. The main reason for this development is that the stealthy nature of VMI has gained relevance as it can assist security experts in dynamically analyzing the behavior of split-personality malware that can choose to behave differently if it notices that it is being analyzed [CAM+08].

In the past, x86 was the dominant player in the server world and an exceedingly attractive target for malware and exploitation; this is where VMI was most needed. Yet, IoT, wearables, mobile devices, the growing demand for ARM in the server market [Ama20, The20b], and Apple's transition to the ARM architecture [AA20] has created renewed emphasis on developing VMI systems for ARM. Even though stealthy VMI has proven itself perfectly suitable for malware analysis on Intel, it often lacks the foundation required to be equally effective on ARM. At this point, the question arises to what extent we can leverage the ARM architecture to improve state-of-the-art dynamic binary analysis frameworks through stealthy VMI techniques on the ARM architecture (Q1).

We have observed that virtualization-based analysis frameworks build upon fundamental techniques that allow to *intercept* and *single-step* guest OSes [DZX13, GVJ14, LMP+14]. To observe the guest's execution, one can apply both *invasive* [DZX13, GVJ14, LMP+14] and *non-invasive* [DRSL08, Lit08, PSE11] approaches. To assist security experts in analyzing sophisticated malware, both approaches must be invisible to the guest (Sec. 2.3.1). Although non-invasive approaches are inherently stealthy, in-guest memory or register artifacts used by invasive approaches to intercept the VM's control-flow must be explicitly hidden. For instance, an adversary can use the finite number of hardware breakpoint registers to reveal the analysis framework. While Intel as well as the AArch64 execution state of ARMv8 CPUs (Sec. 2.2.1) allow to hide memory artifacts by marking memory pages as *execute-only*, second level translation tables of both the AArch32 execution state of ARMv8 and the ARMv7 architecture prohibit *execute-only* memory and thus impede stealthy VMI.

Besides, to transparently single-step guest OSes on Intel CPUs, the Monitor Trap Flag (`MTF`) can be used. As a matter of fact, `MTF` is part of Intel's virtualization extensions and inaccessible to the guest. Sadly, this feature is not supported by ARM. In fact, it is unfeasible to single-step the guest in a stealthy way by relying solely on the hardware capabilities. While previous efforts employ VMI on ARM [YY12, GVJ14, TKFC15], none of them achieves a stealthy solution against attackers with root privileges. Emulation presents a potential workaround, but is known for being imperfect [Fer07, Xen16b, Xen16a].

In this chapter, we closely examine the ARM architecture to identify shortcomings and develop novel techniques necessary for effective virtualization-assisted dynamic binary, or rather malware analysis. Specifically, we explore novel directions of VMI primitives, which empower stealthy monitoring of guest OSes with multiple *virtual CPUs* (vCPUs) on both AArch32 and AArch64 without resorting to emulation. First, we introduce an alternative method on placing breakpoints; instead of using hardware or software breakpoints that either leak information about the analysis framework or require logic that distinguishes between breakpoints set by the analysis system and the guest—thus increasing the performance overhead—we expand the idea that was first presented in SPROBES [GVJ14]. As such, we place Secure Monitor Call (`SMC`) instructions into the guest kernel to intercept the VM and redirect the control to the hypervisor. By injecting only two `SMC` instructions, we enable single-stepping without using the hardware-intended approach, which can reveal the monitor. In parallel, we implement a system, which facilitates an external monitor that applies second level address translation (SLAT) tables to define and dynamically switch among different guest-physical memory views. In this context, we introduce our extensions to the Xen Project hypervisor [BDF+03, Lin20a] on ARM, called *alternate* `p2m` (`altp2m`). We highlight that even though we decided to extend the Xen Project hypervisor, the introduced primitives are not bound to any VMM. In fact, we can consider leveraging the stealthy deployment strategy of WhiteRabbit (Chap. 4) in order to dynamically mount the analysis techniques discussed this chapter.

The above methodologies suffice for stealthy analysis on AArch64. Yet, to hide from malware on AArch32, which lacks *execute-only* memory even through SLAT, we consolidate the aforementioned techniques along with the Translation Lookaside Buffer (TLB). In

detail, we take control over the TLB organization on AArch32 by leveraging `altp2m` to establish a stealthy guest monitoring approach. This approach employs `altp2m` to carefully de-synchronize the TLB organization to effectively hide code pages in the guest's memory. This allows us to maintain different mappings of the same guest physical frame in the *instruction* and *data* TLB and thus to effectively hide code pages in guest memory [Tor14].

Finally, we extend the dynamic malware analysis framework, DRAKVUF [LMP+14], with the capabilities of the above primitives to empower ARM for stealthy VMI.[1] In fact, we leverage DRAKVUF to evaluate the performance and effectiveness of our VMI primitives on AArch32 and AArch64. Overall, we believe that our work constitutes an efficient and robust building block that is able to introduce stealthy VMI to the ARM architecture.

Note: Parts of this chapter have been published in [PLM+18]. We have open sourced the code developed as part of this project. In fact, to allow other researchers to reproduce our results, we have participated in the conference's artifact evaluation.[1] The published research paper has received an *Outstanding Paper Award*.

[1] DRAKVUF on ARM: `https://github.com/drakvuf-on-arm/drakvuf-on-arm`

5.1 The Need for Alternative Monitoring Primitives on ARM

Split-personality malware frequently employs anti-virtualization techniques [CAM+08, BCK+10] to reveal and evade introspection. Even though defenders could put a lot of effort into an attempt to make the VM almost indistinguishable from real hardware, a system that achieves perfect VM *transparency* is still unfeasible in practice [GAWF07] (Sec. 2.3.1). Recently, as we have learned in Chap. 4, we have observed an increasing trend toward system consolidation through virtualization. This trend renders the goal of VM transparency obsolete; a virtualized system does not necessarily indicate that its sole purpose is malware analysis. Therefore, it makes no sense for attackers to exclude virtual environments. Nevertheless, malware can still detect VM-based analysis systems, through in-guest artifacts, ranging from guest-accessible memory to register contents. Consequently, cloaking remains an open question and emphasizes the need for stealthy monitoring.

Contrary to x86, ARM does not foresee hardware capabilities that are essential for *stealthy* malware analysis. In fact, this is one of the reasons why existing VMI approaches for x86 cannot be similarly applied to ARM. In particular, ARM is not capable of hiding artifacts that are involved in single-stepping guest VMs. Additionally, ARMv7 complicates hiding in-guest code instrumentation, as it lacks the capability of granting *execute-only* permissions to code pages. While both points can be addressed through emulation techniques, we choose to avoid emulation, as it is known for being imperfect [Fer07, Xen16b, Xen16a]. In this section, we extend the introduced ARM architecture (Sec. 2.2) to highlight its limitations in regard to stealthy VMI. We particularly amplify upon the selected architectural components that are relevant for this chapter.

5.1.1 Debug Exceptions

The ARM architecture offers a set of debug registers, which can be configured to set breakpoints and watchpoints, and to single-step individual instructions. Specifically, ARM allows to configure the debug registers to generate debug events. These events, in turn, generate debug exceptions which must be handled in dedicated exception handler routines that are typically set up by debuggers.

Both AArch32 and AArch64 support up to 16 configurable breakpoints. Every breakpoint can be set by means of the Breakpoint Control Register `DBGBCR` in conjunction with one of the Breakpoint Value Registers `DBGBVR`. Respectively, ARM supports up to 16 watchpoints, which function in a similar way. In the simplest case, a set breakpoint or watchpoint holds an instruction address, that generates an associated debug event on every instruction or data fetch. On top of that, ARM features software breakpoint instructions, the execution of which causes the CPU to generate the respective debug events. For instance, on AArch64, upon executing a software breakpoint instruction, the system immediately generates a Breakpoint Instruction exception, which interrupts the most recent execution and transfers the control-flow to the registered exception handler.

To single-step hit breakpoints, a monitor can configure DBGBCR to mismatch the breakpoint address in one of the DBGBVR registers: since addresses of instructions that follow a hit breakpoint do not match the address in DBGBVR, the execution of every following instruction will cause the CPU to generate a debug event. Alternatively, AArch64 allows to generate Software Step exceptions by setting the SS bit of the Monitor Debug System Control, MDSCR_EL1, and Saved Program Status Register, SPSR, of the target exception level. For instance, to single-step a hit breakpoint in EL1 the monitor must set the MDSCR_EL1.SS and SPSR_EL1.SS bits. After returning to the trapped instruction, the SPSR will be written to the process state PSTATE register in EL1. As a consequence, the CPU will execute the next instruction and generate a Software Step exception.

To prevent a potential disclosure of the analysis system, the VMM can intercept (and emulate) the guest's accesses to debug registers and hence cover, e.g., set breakpoints controlled by the VMM. Yet, we highlight that adversaries can use the finite number of breakpoint and watchpoint registers as side channel information to reveal the analysis framework. Also, in-guest debugging cannot be perfectly emulated. For example, the KVM hypervisor implements VMM-based debugging on ARM with the restriction that the guest will be unable to use these features concurrently. Thus, hardware breakpoints and watchpoints, as well as single-stepping through breakpoint mismatching—the only way to single-step guest's on AArch32—are not suited for stealthy VMI.

Unfortunately, Software Step exceptions on AArch64 are also visible to guest OSes. The VMM can intercept accesses to MDSCR_EL1 and hide the SS bit. Also, ARM forbids direct access to the PSTATE.SS bit in all exception levels, which complicates a discovery of analysis systems. Still, an adversary with control over the guest's exception handlers in EL1 (kernel space) can reveal the analysis by provoking an interrupt from EL1 that traps as well to EL1 (Sec. 2.2.1): in the exception handler, the PSTATE holding the set SS bit will be written to SPSR_EL1 which in turn is accessible. We have validated this behavior as part of our evaluation (Sec. 5.4.4). Since accesses to SPSR_EL1 cannot be intercepted, the VMM would need to trap and emulate every instruction in the exception handlers to cloak the analysis. This, however, is not a good alternative as the overhead of handling exceptions would rise enormously. As such, we are in need for *stealthy* single-stepping alternatives, upon which we place great emphasis in this chapter.

5.1.2 Translation Lookaside Buffer

To counter the lack of *execute-only* memory on AArch32, we shift our focus toward the TLB organization. Virtual memory address translation entails high performance overhead. The reason for this is that the associated page tables reside in main memory. To increase performance, the TLB buffers the most recent guest-virtual to guest-physical address (and guest-physical to machine-physical) translation results. The TLB organization on x86 and ARM evolved to a split TLB architecture. A split TLB separates the TLB into two disjoint sets comprising the instruction TLB (iTLB) and data TLB (dTLB); the iTLB caches translated instruction fetches, whereas the dTLB holds translated data fetches. To further speed up

memory translation, the TLB organization adds a superior caching level, which serves as a victim cache for both the instruction and data TLB. On ARM, this cache is called unified TLB (uniTLB); it is comparable to the shared TLB (sTLB) on x86. The uniTLB holds evicted entries from the iTLB and dTLB and is first consulted before walking the page tables.

To minimize TLB maintenance, the TLBs are associated or tagged with an identifier, which organizes the TLB entries based on a specific context. This means TLB entries with the same Address Space Identifier tag refer to a specific process. Similarly, entries tagged with the same Virtual Machine Identifier (`VMID`) refer to a specific VM or rather to a specific second level translation table (the `VMID` is part of the Virtualization Translation Table Base Register `VTTBR`). As such, the CPU does not need to flush the TLBs on context switches, which significantly increases the overall system performance.

5.2 Threat Model

Our assumptions regarding the adversarial capabilities closely resemble the threat model in Sec. 4.1. In a similar way, we base and extend the assumptions of the *defensive* attacker strategy in Sec. 3.2. Previously, the attacker has focused on uncovering traces of WhiteRabbit, once it has been dynamically deployed (Chap. 4). Yet, WhiteRabbit offers a generic VMI interface, which can tolerate and provide services to non-stealthy VMI tools. In other words, while the attacker cannot directly uncover WhiteRabbit, she could indirectly expose the analysis system through artifacts that were left behind by careless VMI tools. In addition to our previous assumptions, to amplify upon the stealthy character of the introduced analysis techniques in this chapter, we assume an adversary who specifically aims to reveal in-guest analysis artifacts. Note that we can combine the stealthy deployment strategy of WhiteRabbit with the stealthy analysis techniques in this chapter.

We assume an adversary with root privileges, who possesses full control of the guest VM on ARM; she can access all security-relevant parts of the OS, including the guest kernel and the configured exception vectors. Similar to our previous assumptions, we assume the attacker is indifferent to virtualized systems. Yet, she will abort her operation in case of disclosure of an analysis framework. Additionally, she can employ anti-virtualization techniques, such as carving the guest's memory for artifacts that may reveal the presence of virtualization-based analysis frameworks. These artifacts comprise instructions, such as software breakpoints and hypercalls, which are capable of intercepting and redirecting the guest's execution to the VMM. Further, she can inspect the OS for agents in form of processes or kernel modules. We trust that the adversary cannot modify the OS before we have set up our VMI-based analysis framework. That is, she cannot manipulate any security-critical vectors without us noticing it.

We highlight that throughout this chapter, we consider that the attacker can analyze in-guest accessible registers. Hence she aims to reveal analysis frameworks, which utilize debug registers (Sec. 5.1.1), e.g., to configure hardware breakpoints, watchpoints, and the single-stepping mechanism. The attacker understands that although access to these registers can be intercepted and falsified by a VMM-based analysis framework, the VMM will not be able to cloak the resulting side effects. For instance, if the attacker is being emulated, she has the necessary means to discover it, due to imperfect emulation. The same applies to register contents that spill into registers, access to which cannot be trapped by the VMM. Note that we apply this threat model to the *adore-ng* rootkit to simulate a realistic attacker in our evaluation (Sec. 5.4.4).

5.3 Guest Kernel Monitoring Primitives

Malware can reveal analysis systems that use standard debugging techniques and change its behavior [CAM+08, BCK+10]. Thus, non-stealthy analysis can cause false observations. We identify the requirements, which are necessary to facilitate stealthy monitoring:

❶ a mechanism to *intercept* the guest in `EL1` (guest kernel);

❷ a *single-stepping* mechanism that cannot be discovered through in-guest artifacts; and

❸ *execute-only* memory.

Even though the Intel architecture meets the requirements ❶ – ❸, unfortunately, the ARM architecture does not foresee the necessary means for a stealthy single-stepping mechanism (Sec. 5.1). On AArch32 the finite number of hardware breakpoints helps the attacker to infer the presence of an analysis framework; on AArch64, the attacker can spill the set `PSTATE.SS` bit into the `SPSR_EL1` register to uncover the analysis framework. Besides, AArch32 lacks *execute-only* memory (Sec. 5.1). As such, both architectures do not meet the requirement ❷, whereas AArch32 additionally fails to comply with the requirement ❸. Consequently, many prevalent VMI tools cannot use the hardware features on the ARM architecture to become equally effective as on Intel.

To tackle these shortcomings, we present novel VMI primitives that facilitate stealthy monitoring of VMs. Instead of relying on the intended hardware mechanisms, we apply the system's virtualization extensions in a new way to empower ARM for stealthy VMI. Specifically, we utilize the virtualization extensions to intercept the guest kernel at arbitrary locations (aka. *tap points* [PCSL08]) and leverage such tap points to single-step trapped instructions in a stealthy way. Next, we extend the Xen Project hypervisor [BDF+03, Lin20a] to control the guest's physical memory, which presents the foundation for stealthy monitoring on ARM. Finally, we combine these primitives to set the ground for stealthy VMI on ARM. Fig. 5.1 illustrates an overall architecture of our system whose components are described in this section. In the following, we present primitives that, when combined, form stealthy monitoring systems on both AArch64 and AArch32.

5.3.1 Implementing Kernel Tap Points

A monitor can leverage second level address translation (SLAT) to intercept the guest's execution at arbitrary locations. By withdrawing the `execute` memory access permission from code pages containing functions of interest, the monitor can redirect the guest's execution in `EL1` and `EL0` to the VMM (i.e., in `EL2`). Yet, this coarse-grained method incurs a high overhead: execution of code that is irrelevant for tracing on the page holding the target instruction would trap into the VMM. Instead, it is more desirable to monitor only the fact that the guest has executed a specific function. This demand can be met by software breakpoints. Software breakpoint instructions (Sec. 5.1.1) lend themselves as

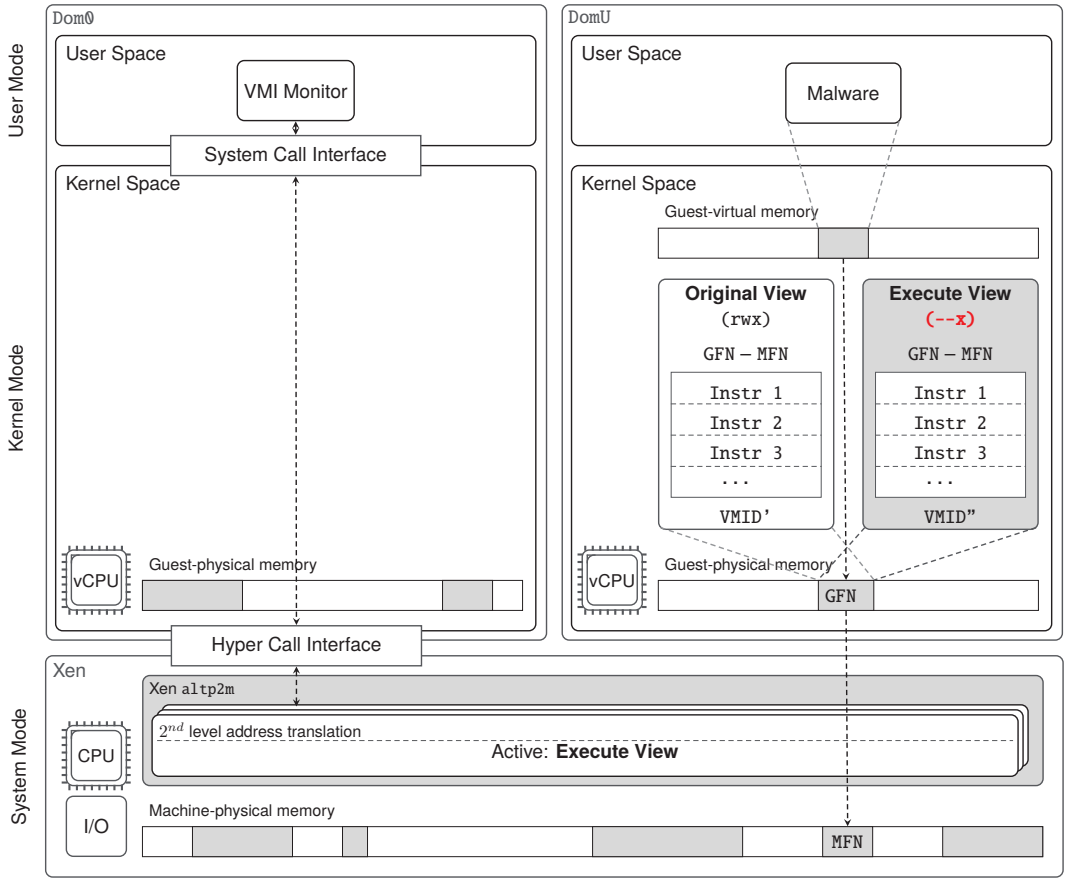
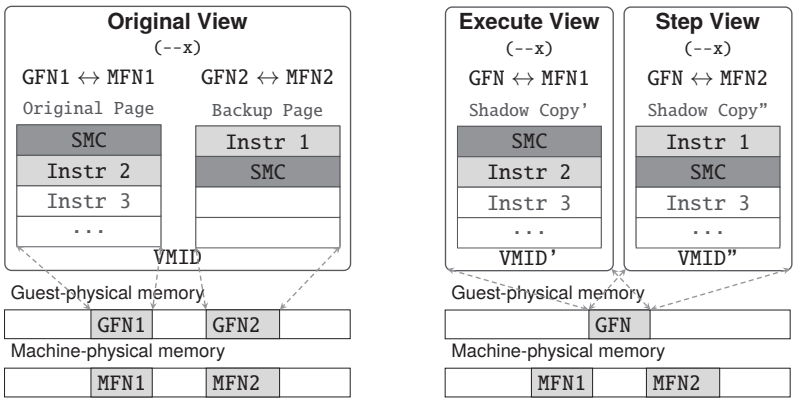


Figure 5.1: Xen `altp2m` enables a monitor in the privileged domain *Dom0* to manage different memory views of *DomU*. While the *execute-view* maps the target guest frame as *execute-only*, the permissions of the *original-view* remain unchanged.

the instruction of choice to implement *tap points* (❶). Yet, there exist other instructions that can be similarly configured to trap to the VMM and have additional properties that make them more desirable. For instance, the Secure Monitor Call (`SMC`) instruction is a great alternative. Similar to software breakpoint instructions, the `SMC` instruction can be configured to trap directly to the VMM. Thus we can utilize `SMC` instructions as triggers to intercept and redirect the control-flow of the guest OS to the VMM (❶).

One of the main benefits of using the `SMC` instruction is that the guest is architecturally unable to subscribe to `SMC` traps; contrary to software breakpoints, the generated Secure Monitor Call exceptions can only be directed to the TrustZone or to the VMM. This property reduces the complexity of the monitor, as the execution of an `SMC` instruction never has to be re-injected into the guest. A limitation of the `SMC` in place of a software breakpoint is that it can only be executed in `EL1`, that is the guest kernel.

Even though we can use the `SMC` instruction to implement tap points in the guest kernel (❶), achieving *stealthy* single-stepping without architectural support presents a great



(a) Without Xen `altp2m`. (b) With Xen `altp2m`.

Figure 5.2: Upon executing the first `SMC` instruction in the original page, the VMM redirects the control-flow either to the backup page (a) or switches to the *step-view* (b) to single-step `Instr 1`. In both cases, the pages are marked as *execute-only*.

challenge. Without hardware support for stealthy single-stepping we seem to have reached an impasse on how to resume the execution of the guest kernel, without losing control over the guest's control-flow. If we remove the tap point to allow the trapped vCPU to continue its execution, an additional event needs to be triggered immediately after executing the first instruction, so that we can place the tap point back. This event will normally be generated by a single-step exception. Further, in multi-vCPU domains, removing the tap point from memory is critical, as it may introduce a race condition: while removing the tap point, other vCPUs must not fetch the instruction from the same location.

5.3.2 Novel Single-Stepping Mechanism

The ARM architecture has an advantage over its x86 counterparts that we can leverage for a novel *single-stepping* scheme, without using the single-stepping feature of the CPU: ARM implements a fixed-width ISA. On x86, software breakpoints cannot be placed at arbitrary locations, as you may end up overwriting a part of a large instruction. On ARM, we can determine the position of the next instruction; depending on the execution mode, the width of instructions is known beforehand. Thus, we can locate instruction boundaries in memory without having to rely on a disassembler.

To illustrate how to utilize the fixed-width ISA for single-stepping, let us consider a scenario, in which we run a VM with a single vCPU. An external monitor with access to debug information of the target kernel, such as the *System.map* file on Linux, can determine the location of system call handling kernel functions. By reading the first two instructions from the prologue of the target kernel function into a backup buffer and then overwriting the function's first instruction with an `SMC`, the monitor will be able to intercept the guest kernel (❶) on execution of the marked kernel function. As soon as the guest kernel executes

the instrumented SMC instruction, the monitor will intercept the guest's execution. At this point, the monitor can place the temporarily saved *original* first instruction back into memory and replace the immediately following instruction with a second SMC. When the VMM resumes the VM, the guest executes the original instruction followed by the second SMC. When the guest traps on the second SMC, the monitor can restore the original, first SMC without losing control over the guest kernel and hence conclude single-stepping of the first instruction in the system call handler. On AArch64, we achieve stealthy single-stepping of the guest (❷) by configuring the instrumented page, which holds the system call handler, as execute-only (❸); reads and writes will trap into the VMM.

To achieve multi-vCPU safety, we can store the first instruction at a location already mapped as executable, but unused at run-time. Here, we can safely place the second SMC after the stored instruction. The exact location of this memory is flexible as we only need space for two instructions per monitored location. For this, we can leverage known memory holes in the Linux kernel, such as the memory immediately following the kernel. For simplicity, we dedicate an entire page, *backup page*, for these two instructions (Fig. 5.2(a)). In the figure, the *original-view* represents the physical memory that is made visible to the guest through SLAT. On execution of the first SMC in the system call handler, we point the trapped vCPU's Program Counter (PC) to the *backup page* holding the original first instruction without performing any further modifications. Once the second SMC in the *backup page* is executed, we will point the PC back to the instruction following the first SMC in the *original page*. While the above focuses on single-stepping the first instruction of system call handlers, we can apply the same approach for arbitrary regions in the guest kernel. This however, must foresee corner-cases, such as function returns and branches, and thus requires the monitor to compute the target address.

5.3.3 Xen `altp2m` on ARM

Due to the architectural differences between x86 and ARM, we cannot apply existing VMI solutions that target x86 to the ARM architecture. To nevertheless adapt existing malware analysis tools that rely on the requirements ❶, ❷, and ❸ to the ARM architecture, we mimic the behavior of an effective approach for Intel, namely Xen *alternate p2m* subsystem (short `altp2m`). Without divulging the details of Xen `altp2m` on Intel, which we further expand on in Sec. 6.1, the developers of Xen `altp2m` have originally designed and implemented this subsystem for the exclusive use on Intel. On this architecture, a VM's memory view can be directly associated with an Extended Page Table (EPT) represented by the EPT pointer (EPTP) in the hardware defined data structure Virtual Machine Control Structure (VMCS) (Sec. 2.2.2.1). The VMCS has capacity for up to 512 EPTPs (i.e., memory views) among which the system can be instructed to switch dynamically. To the best of our knowledge, Xen `altp2m` is the first public implementation that makes use of this CPU feature. This makes Xen `altp2m` a unique tool for virtual machine introspection [Len16].

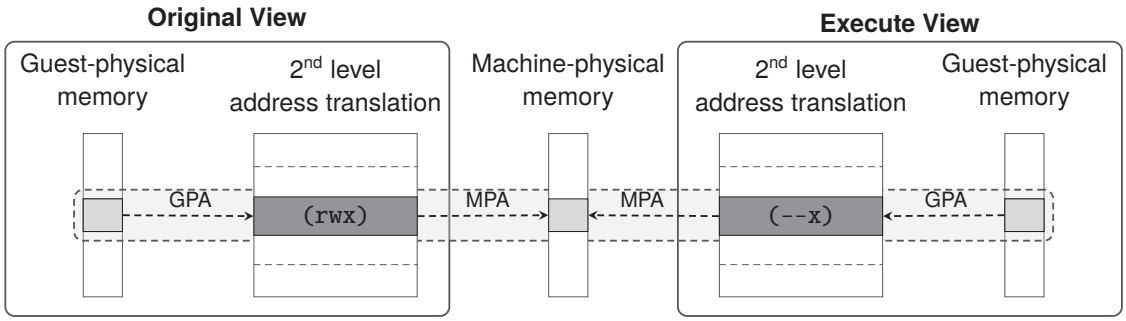


Figure 5.3: The *execute-view* maps the target guest frame as *execute-only*; the *original-view* grants the same guest frame `read`, `write`, and `execute` access permissions.

We implement the `altp2m` subsystem for ARM upon the Xen `p2m` subsystem. Xen `p2m` stands for *physical to machine* and leverages SLAT to manage and isolate memory between guest domains (Xen's notion for VMs) and the VMM. On ARM the Virtualization Translation Table Base Register (`VTTBR`) holds the base address of SLAT tables, similarly to the EPTP on Intel. Xen `p2m` maintains only one `VTTBR`. As such, the `p2m` subsystem maintains a *single* view of the guest's physical memory, even for VMs with multiple vCPUs. On the other hand, Xen `altp2m` on ARM allows to dynamically define and switch among *different* `VTTBR`s per domain and vCPU. The interaction with the `altp2m` interface takes place through dedicated hypercalls, called `HVMOP`s. These facilitate the privileged domain *Dom0* to create, switch, and destroy individual memory views that are then applied to unprivileged domains *DomU* (Fig. 5.1). In addition, the `altp2m` interface allows to define memory access permissions of individual guest-physical page frames per view and also remap individual guest frames to different machine frames.

VMI tools leverage SLAT to control the access permissions of the guest's physical memory [DZX13, LMP+14] (Sec. 2.2.3). When the guest traps into the VMM due to a memory access violation, the access permissions of the associated entries must be temporarily relaxed; the VMM must grant the required permission so that the guest can continue. Yet, relaxing permissions in this ubiquitous view may allow one of the remaining vCPUs to access the targeted memory without notifying the VMM. One solution is to pause remaining vCPUs while single-stepping the trapped vCPU. This, however, imposes severe performance degradation. Also, the lack of stealthy single-stepping on ARM makes VMI-tools susceptible to disclosure. Xen `altp2m` solves such race conditions by maintaining *different* views of the guest's physical memory (Fig. 5.1). Instead of changing permissions of a single memory view at run-time, `altp2m` allows to allocate a set of views beforehand. This way, a monitor can individually assign a specific memory view to each vCPU of *DomU*. As such, for instance on memory access violations, VMI-tools can switch the view of the affected vCPU to a less restrictive view, instead of explicitly relaxing permissions of the view that led to the trap; switching views is as simple as switching the domain's `VTTBR`.

Let us depict a scenario in which we monitor writes to critical regions (e.g., exception vectors) in a multi-vCPU domain. This scenario assembles the architecture in Fig. 5.1. A monitor in *Dom0* allocates two distinct guest memory views that can be applied to each vCPU in *DomU*. The monitor grants all guest frames of the critical region *execute-only* permissions in the *execute-view*. The same guest frames keep their original permissions in the *original-view* (Fig. 5.3); write attempts to the critical page by vCPUs with an active *execute-view* generate memory access violations. We configure this behavior as default on all vCPUs. On a potentially malicious write attempt, instead of relaxing permissions, the monitor switches the view of the trapped vCPU to the less restrictive *original-view*. This allows us to record the event, satisfy the write request, and avoid the target application to become suspicious. We do not relax permissions of other vCPUs and thus avoid race conditions. To ensure that the monitor regains control immediately after the write, we single-step the trapped vCPU and switch back to the *execute-view*.

To further highlight the potential of Xen `altp2m`, we combine the single-stepping scheme in Sec. 5.3.2 with Xen `altp2m`. For every instruction to be single-stepped, an external monitor has to increase the guest's physical memory by two additional pages. This allows it to create two shadow-copies of the page holding the original instruction (we need two additional copies if we would like to satisfy code integrity checks that can be redirected to a view pointing to the original page). That is, similar to the above scenario, we allocate two additional guest memory views: the *execute-view* holds the first duplicate, *shadow-copy'*, while the *step-view* maps the *shadow-copy"* (Fig. 5.2(b)). We replace the target instruction in the *execute-view* with a privileged `SMC` instruction. Then, instead of allocating a backup page in the same memory view, we replace the second instruction of the same function in the *shadow-copy"*. As such, upon execution of the first `SMC` in the *execute-view*, the monitor can switch to the *step-view* without further adjustment. Finally, upon the execution of the target instruction, the execution of the adjacent `SMC` instruction in *step-view* traps again into the VMM, where we return to the *execute-view* and complete single-stepping.

These scenarios demonstrate the potential of Xen `altp2m`, enforcing memory restrictions through SLAT. The guest has no access to SLAT tables, as they reside in VMM's memory. Thus, such memory restrictions are stealthy. In fact, we meet the requirements ❶ – ❸ and hence establish a foundation for *stealthy* VMI on AArch64.

5.3.4 Splitting the TLBs

Sadly, AArch32 is not capable of enforcing *execute-only* pages; every code page has to be both *readable* and *executable*, or instruction fetches will fail. Thus, while requirement ❶ and ❷ (partially) apply to AArch32, the requirement ❸ is not covered. Therefore, we cannot assure stealthy operation of Xen `altp2m` without further actions on AArch32. To overcome this limitation we explore the TLB organization on ARM (Sec. 5.1.2) and implement a system, which we refer to as *split-TLB* that satisfies our requirements.[2]

[2]The term *split-TLB* first appeared in the context of processor architectures. In this chapter, we use this term also as a substitute for the de-synchronized TLB organization.

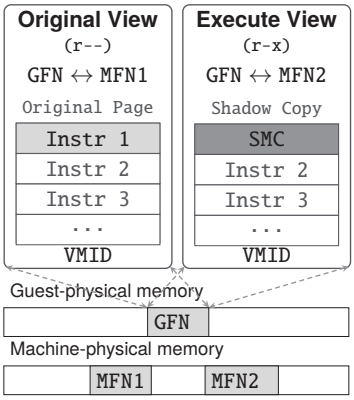


Figure 5.4: The *original-view* translates the guest frame to `MFN1`, while the *execute-view* translates the same guest frame to `MFN2`. Both views are tagged with the same `VMID`.

ARM implements TLB-tagging to isolate translations tagged with different `VMID`s. We made a joint decision with Xen maintainers that unique `VMID`s will be assigned to each `altp2m` view. Contrary to x86, on ARM a TLB tag corresponds to a specific *memory view* instead of a *vCPU*; by switching `altp2m` views we activate the associated `VMID`, without having to flush differently tagged mappings of same guest-physical memory. On the other hand, if different `altp2m` views shared a `VMID`, the guest would be susceptible to using stalled translations in the TLB, even if the active view contained the most recent mappings in memory. We choose to employ this architectural feature to hide modified code pages from data fetches and thus mimic *execute-only* memory on AArch32. As such, we extend the `altp2m` interface to pair the `VMID`s of `altp2m` views to de-synchronize the physically separated *iTLB* and *dTLB*.

To cause an inconsistent state in the TLBs that we require for hiding code pages, we prime the iTLB so that it holds guest frame mappings that translate to different machine frames than those cached in the dTLB. That is, we require a mechanism that allows us to translate one guest frame to two physically different machine frames; only one of both mappings will be exclusively cached either in the iTLB or dTLB. To achieve this effect, first, we duplicate the page with the instruction to be monitored and replace it with an `SMC` instruction in the *shadow-copy* without modifying the *original page*. Then, we prepare two `altp2m` views and map both pages according to Fig. 5.4. It is essential that both memory views are tagged with the same `VMID`; the system will ignore the primed iTLB entry, if it switches to a memory view with a different `VMID`. We grant the *original page read-only* permissions. We withdraw write permissions from the *original page* in the *original-view* to intercept write attempts. This allows us to monitor any change to the original page and propagate the modifications to the shadow-copy as required. Since AArch32 does not support *execute-only* mappings, we grant the *shadow-copy* `read` and `execute` memory access permissions. Also, we withdraw the `execute` permission from all other mappings in the *execute-view*, as to limit the execution in this view to the page of interest (Fig. 5.5).

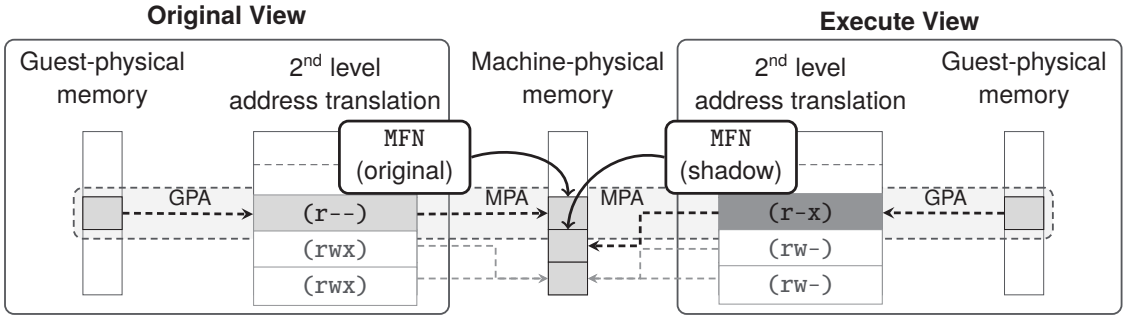


Figure 5.5: The *original-view* maps the target guest frame to the original machine frame; the *execute-view* maps the same guest frame to the shadow-copy.

We configure the *original-view* to be active by default. On the first execution of the function to be monitored, the guest hands over control to the VMM: the instruction fetch violates the permissions of the *read-only* mapping. As such, the translation result does not get cached in the TLBs. The monitor leverages this architectural property to intercept the guest upon permission violation and switch to the *execute-view*, which grants execution access of the requested guest frame. This time, upon the successful SLAT table walk using the *execute-view*, the translation mechanism populates the iTLB with the machine frame that is associated with the *execute-view* (i.e., `MFN2` in Fig. 5.4). Consequently, further instruction fetches from the page in question will directly consult the primed iTLB entry until it gets evicted. When the primed iTLB entry gets evicted it will need to be primed again. Upon execution of the `SMC` in the *execute-view*, the monitor can single-step the original instruction as described in the Sections 5.3.2 and 5.3.3. After single-stepping the monitored instruction, the monitor switches back to the *original-view*.

The setup configures two views that map one guest frame to two machine-physical frames with different access permissions. By priming the iTLB, we cause the system to fetch the target instruction from the *execute-view*. At the same time, reads from the same addresses consult the *original-view*. As the iTLB and dTLB hold mappings from two different views, the primed system does not need the VMM to switch the views. This setup satisfies reads initiated, for example, by integrity checkers. At the same time, it transparently causes the guest to execute the `SMC` instruction in the *shadow-copy* of the *original page* (❸). Thus, by combining the monitoring primitives introduced in this section with the discussed TLB de-synchronization technique, we manage to meet the requirements ❶–❸ on AArch32. In addition, since this configuration causes the VM to trap to the VMM only for the purpose of priming the TLBs, or when executing the instrumented `SMC` instructions, the split-TLB strategy incurs only minimal overhead. Nevertheless, this setup entails a limitation that we discuss in detail in Sec. 5.5.2.

5.4 Evaluation

To evaluate our work, we have implemented the discussed Xen `altp2m` subsystem and ported LibVMI [Pay12, Lib20] and the dynamic binary (and malware) analysis framework DRAKVUF [LMP+14] to the ARM architecture.[1] We have equipped DRAKVUF with the presented VMI-primitives to assemble the foundation for stealthy guest kernel monitoring on AArch32 and AArch64. This allowed us to assess both the effectiveness and the induced performance penalty of the introduced primitives.

5.4.1 System Setup

Our system setup comprises an external monitor (i.e., DRAKVUF) inside of the privileged domain `Dom0` on top of the Xen hypervisor v4.11. DRAKVUF traces system calls that are executed in the unprivileged domain `DomU`. Both domains run the Linux kernel v4.15. The host comprises an 8-core 1.2GHz ARM Cortex-A53 CPU, and 1GB of RAM available to both the privileged domain `Dom0` and the unprivileged domain `DomU`. Even though we have performed all measurements on AArch64, the setup can be equally applied to AArch32.

DRAKVUF uses OS-profiles, with exported functions and data structure information, that are statically generated by Rekall [Rek20] to locate system calls and set tap points in the prologue of each system call handler in the guest's kernel memory. This way, it establishes the means to intercept and monitor the guest's kernel behavior. Unfortunately, Rekall lacks the ability to generate profiles for Linux kernels compiled for AArch64. To accommodate this issue, we have implemented and open-sourced a custom Rekall profile generator to gather the relevant AArch64 system call and kernel data structure information.[3]

As part of our evaluation, DRAKVUF leverages our implementation of the Xen `altp2m` subsystem (Sec. 5.3.3) to dynamically create and switch among different guest memory views on ARM;[4] it uses the `altp2m` interface to communicate with the introduced subsystem from `Dom0`. We employ `altp2m` in combination with the discussed VMI-primitives that meet our requirements ❶ – ❸ to stealthy monitor every system call on AArch64 (Sec. 5.3). In total, our system setup has monitored 340 different system calls, which are distributed across 111 different 4 KB memory pages. In addition, we have armed `altp2m` with the ability to take control over the system's TLB organization to assess the primitives for stealthy monitoring on AArch32 (Sec. 5.3.4).

[3] Rekall profile generator for AArch64:
`https://github.com/drakvuf-on-arm/rekall-profile-generator`

[4] Xen `altp2m` on ARM: `https://github.com/drakvuf-on-arm/xen/tree/arm-altp2m-drakvuf`

5.4.2 DRAKVUF on ARM

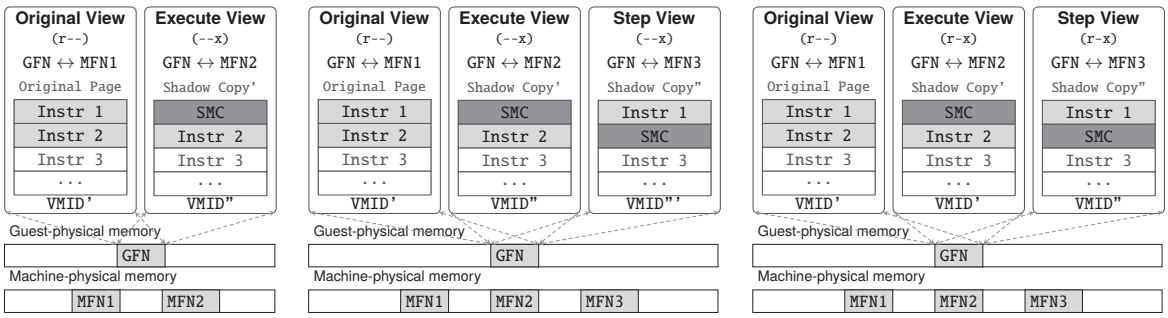
To assess the presented VMI-primitives for the ARM architecture (Sec. 5.3), we have combined them with DRAKVUF. The established foundation for DRAKVUF enabled thereby stealthy dynamic malware analysis on ARM. In addition, to demonstrate that our proposed single-stepping primitive (by using both the synchronized and the de-synchronized TLB configuration) can keep up with the performance of the hardware-supported approach, we have implemented a conventional, non-stealthy single-stepping mechanism for Xen in the way it is specified by the AArch64 specification [Arm20]; this single-stepping alternative leverages dedicated hardware capabilities controlled through the `PSTATE.SS` bit on AArch64 (Sec. 5.1.1).

That is, within the scope of our evaluation, we have extended Xen, LibVMI, and DRAKVUF in such a way that allows us to employ the VMI-primitives for placing arbitrary *tap points* in the guest's kernel memory and protecting these through our *Xen* `altp2m` subsystem implementation. At the same time, we can combine these primitives with one of the following three different single-stepping approaches:

- *Hardware-SS*: a non-stealthy single-stepping implementation that leverages hardware capabilities of AArch64 (Sec. 5.1.1).
- *Double-SMC-SS*: our stealthy method that leverages two `SMC` instructions protected by Xen `altp2m` (Sec. 5.3.3).
- *Split-TLB-SS*: an approach that additionally de-synchronizes the TLB organization to make up for the lack of execute-only memory (❸) on AArch32 (Sec. 5.3.4).

Independent of the applied single-stepping method, DRAKVUF utilizes `SMC` instructions to set tap points in the prologue of each system call handler and leverages Xen `altp2m` to protect them. In other words, in the face of the evaluation, we have applied one of the three single-stepping flavors through DRAKVUF to step over the trapped `SMC` instructions. More precisely, we have leveraged DRAKVUF to duplicate all pages, which hold system calls, such that we can configure them to be monitored (Fig. 5.6). Then, we have allocated two guest memory views, whereas the first view (*original-view*) has mapped the original page and the second view (*execute-view*) has mapped the shadow-copy. We granted the original page in the *original-view read-only* permissions. The permissions of the shadow-copy in the *execute-view* were either *execute-only* on AArch64 or *read-execute* on AArch32. In both *Hardware-SS* and *Double-SMC-SS* setups, it was the *execute-view* that was active by default; *Split-TLB-SS* used the *original-view* as default. Finally, we replaced the first instruction of every system call handling function in the shadow-copy with an `SMC` instruction. Further operation depended on the employed single-stepping method. The following exemplifies the individual steps taken by DRAKVUF to trace every invocation of a system call inside the guest domain in accordance with one of the single-stepping alternatives.

Hardware-SS: To prevent the guest from discovering the instrumented `SMC` instruction in the shadow-copy, we grant *execute-only* access permissions to the memory page that is



(a) Hardware-SS (b) Double-SMC-SS (c) Split-TLB-SS

Figure 5.6: Different views on the guest-physical memory provided by Xen `altp2m`. We utilize each view configuration to establish three single-stepping (SS) alternatives.

mapped in the *execute-view* (Fig. 5.6(a)). This way, the execution of the first instruction of the system call handler in the *execute-view* interrupts the guest kernel execution and hands over control to Xen, which in turn notifies DRAKVUF about the trap. Consequently, DRAKVUF switches back to the *original-view*, single-steps only one instruction (i.e., the original first instruction of the system call handling function) by utilizing the dedicated single-stepping mechanism of the hardware, and continues in the *execute-view*. We can satisfy all read-requests to code pages that are marked *execute-only* in the *execute-view* by switching to the *original-view*. Also, we intercept writes to propagate changes to all views.

Double-SMC-SS: In this context, we use the same *original-view* and *execute-view* configurations as in *Hardware-SS*. In addition, we create a third view, *step-view*, which we use to single-step instructions without relying on the intended hardware capabilities. The *step-view* maps a second copy of the original page, in which we replace the *second* instruction of the system call handler function with a second `SMC` (Fig. 5.6(b)). This way, when we intercept the first `SMC` in the *execute-view*, DRAKVUF switches to the *step-view* to execute the original first instruction and trap at the second `SMC` immediately after. This is the single-stepping scheme, which we have introduced in Sec. 5.3.3. The second `SMC` instruction in the *step-view* facilitates DRAKVUF to switch back to the *execute-view* and continue execution right after the first `SMC`.

Split-TLB-SS: This setup closely resembles the configuration of *Double-SMC-SS*. Yet, since AArch32 lacks *execute-only* memory, we have de-synchronized the system's TLB organization to simulate *execute-only* memory (Sec. 5.3.4). Specifically, we granted *read-execute* memory access permissions to the shadow-copies in the *execute-view* and *step-view* Fig. 5.6(c)). To de-synchronize the entries in the iTLB from the dTLB, we used the same `VMID` for the *original-* and *execute-view*. We primed the iTLB such that instruction fetches from the target page accessed the *execute-view* in the iTLB; data access consulted the *original-view* in the dTLB. Right after executing the first `SMC`, DRAKVUF dynamically switched to the *step-view* to single-step the original first instruction similarly to *Double-SMC-SS*.

Table 5.1: Monitoring overhead of DRAKVUF utilizing the single-stepping primitives *Hardware-SS*, *Double-SMC-SS*, and *Split-TLB-SS*, measured by LMbench v3.0.

	Benchmark	w/o (in *msec*)	Hardware	Double-SMC		Split-TLB	
				Step-View	Backup Page	Step-View	Backup Page
Latency	`syscall()`	0.31	299.05 ×	238.55 ×	269.32 ×	317.68 ×	250.26 ×
	`open()/close()`	5.44	37.25 ×	29.11 ×	35.95 ×	49.57 ×	33.94 ×
	`read()`	0.67	141.14 ×	117.27 ×	125.46 ×	148.27 ×	111.77 ×
	`write()`	0.47	203.32 ×	163.81 ×	178.43 ×	219.62 ×	157.31 ×
	`select()` (500 fds)	28.33	4.40 ×	3.89 ×	4.04 ×	4.39 ×	4.02 ×
	`stat()`	2.63	38.06 ×	30.74 ×	32.43 ×	40.14 ×	31.82 ×
	`fstat()`	0.62	152.57 ×	126.40 ×	135.81 ×	166.97 ×	121.06 ×
	`fork()+execve()`	1383.33	4.38 ×	4.02 ×	4.36 ×	19.29 ×	12.33 ×
	`fork()+exit()`	377.43	2.21 ×	2.09 ×	2.45 ×	15.66 ×	11.20 ×
	`fork()+/bin/sh`	3249.17	3.86 ×	3.59 ×	3.92 ×	16.35 ×	10.54 ×
	`sigaction()`	0.51	186.27 ×	141.18 ×	147.31 ×	174.65 ×	144.58 ×
	Mem read	1745.00	0.97 ×	0.97 ×	1.00 ×	0.99 ×	0.99 ×
	Mem write	4687.67	0.92 ×	0.92 ×	1.00 ×	0.98 ×	0.98 ×
	Signal delivery	4.34	43.70 ×	34.64 ×	35.51 ×	41.01 ×	36.48 ×
	Page fault	1.49	1.15 ×	1.16 ×	1.09 ×	1.28 ×	1.28 ×
	Pipe I/O	12.26	30.34 ×	28.13 ×	34.69 ×	77.94 ×	39.36 ×

In all three configurations, every time a system call trapped into Xen, it has notified DRAKVUF about the event, which monitored the system call for further processing; on every system call, DRAKVUF has intercepted the guest, monitored the event, single-stepped the trapped instruction according to the applied single-stepping alternative, and resumed the guest. As such, during benchmarking, our monitor was overwhelmed with a persistent shower of system calls.

5.4.3 Performance

Automated VMI-based malware analysis strongly affects the overall system performance. As such, the VMI-induced performance overhead must be kept to a minimum. To evaluate the performance overhead of the introduced VMI-primitives, we have conducted two experiments comprising a set of CPU-intensive macro- and micro-benchmarks, during which we used DRAKVUF to monitor every system call that was set off by the benchmarking tools in *DomU*. To solely focus on the monitoring overhead, we have deactivated the output to the console. All results are means over three runs.

In the first experiment, we used DRAKVUF to trace all system calls that were set off by a set of *LMbench v3.0* micro-benchmarks. This allowed us to analyze the induced performance cost on system-software level (Table 5.1). To be more precise, first, we have executed DRAKVUF in combination with the *Hardware-SS* approach that leveraged the AArch64 hardware architecture to single-step protected tap points and determined the overall execution overhead. In this way, we have established a baseline which we

Table 5.2: Monitoring overhead of DRAKVUF utilizing the single-stepping primitives Hardware-SS, Double-SMC-SS, and Split-TLB-SS, measured by Phoronix v7.6.0.

	Benchmark (*unit*)	w/o	Hardware	Double-SMC	Split-TLB
Apps	Gzip (*s*)	416.81	11.99 %	36.30 %	190.12 %
	N-queens (*s*)	779.78	0.00 %	0.00 %	0.00 %
	7-Zip (*MIPS*)	612.67	-0.54 %	0.11 %	10.17 %

have then compared with DRAKVUF's performance using both of our proposed single-stepping variants, namely *Double-SMC-SS* and *Split-TLB-SS*. Besides, as both variants can leverage a *backup page* in the *execute-view* instead of using an additional *step-view* (Fig. 5.2(a)), we examined the performance of both configurations for each single-stepping primitive. Overall, the results show that the performance of *Double-SMC-SS* and *Split-TLB-SS* are close to our baseline implementation for single-stepping and thus present suitable alternatives.

Surprisingly, we have observed that the *Double-SMC-SS* implementation outperforms the setup utilizing *Hardware-SS* in the most cases. One can argue that our implementation that is responsible for managing the hardware-supported single-stepping in Xen requires additional overhead that might result in the observed behavior. However, we believe that the hardware logic behind single-stepping over one instruction requires more time than simply intercepting the execution of an `SMC` instruction.

The performance of the *Split-TLB-SS* approach strongly depends on the number of memory pages are involved in the system call. That is, since we have only a very limited number of iTLB-entries (10 iTLB entries on ARM Cortex-A53), as soon as the system call accesses memory pages that are not yet part of the iTLB, primed iTLB entries might get evicted according to the TLB eviction strategy.

In our second experiment, we have conducted a set of CPU-intensive macro-benchmarks of the *Phoronix Test Suite v7.6.0* and summarized the results in Table 5.2. Unfortunately, due to the limited support on ARM, we were able to conduct only a small number of macro-benchmarks. Please note that the units of our measurements vary. Overall, the collected results suggest that DRAKVUF incurs only limited overhead on the overall system performance and thus is very well suited for efficient malware analysis on ARM.

5.4.4 Effectiveness

To demonstrate the effectiveness of DRAKVUF on ARM, we have set up our system to analyze the *adore-ng* rootkit on ARM. By setting a tap point to trace `kallsyms_lookup_name()`, we identiefied that the rootkit determined the location of the kernel function for kernel hot patching (`aarch64_insn_patch_text()`). In particular, it created kernel hooks required, among others, to hide files, processes, and logs and to communicate with the rootkit. As such, we set another tap point to `aarch64_insn_patch_text()` and hence observed all malicious writes to kernel regions holding sensitive function hooks.

We have further lend the *adore-ng* a split-personality property that was looking for artifacts that could reveal our monitor. When the rootkit had detected our analysis framework, it terminated its operation. With this property, *adore-ng* was able to successfully uncover our system when it leveraged *Hardware-SS* to trace and single-step the rootkit (Sec. 5.1.1). Also, the rookit was able to reveal `SMC` instructions by synchronizing the TLBs, when we applied our *Split-TLB-SS* scheme on AArch32. The exact steps to disclose *Split-TLB-SS* are described in Sec. 5.5.2. Yet, the *Double-SMC-SS* method on AArch64 remained undisclosed. Consequently, as we can use the AArch64 architecture to also trace AArch32 guests, we deem this analysis method stealthy for both architectures.

5.5 Discussion

Even though we provided a strong foundation for stealthy VMI on the ARM architecture, in this section, we discuss alternative tracing methods, review the limitations of the proposed VMI primitives, and amplify upon some of the security concerns entailed with stealthy analysis primitives.

5.5.1 Alternative Tracing Methods

Our approach necessitates two guest context switches to trap on and single-step one instruction. If we limit ourselves to tracing the Linux kernel, we can adapt the functionality of `ftrace` [The20a], a tracing framework for Linux kernel analysis. If we assume a Linux kernel compiled with the `CONFIG_DYNAMIC_FTRACE` parameter, the prologue of white-listed kernel functions will hold a call to a dedicated stub, calls to which allow `ftrace` to record the function call. When tracing is disabled, this stub is filled with `NOP` instructions. We can reuse the call to the function stub in every kernel function by placing an `SMC` instruction to this position and protecting it through Xen `altp2m`. This approach eliminates the need for the second `SMC` (Fig. 5.2), as we do not need to replace and single-step any instructions.

By using this approach, we could reduce the overhead induced by single-stepping instructions. At the same time, this approach limits itself to monitoring only Linux guests with `ftrace` support. Besides, if we prefer to avoid single-stepping instructions, the `ftrace` tracing mechanism will need to be deactivated, which would potentially indicate the analysis framework. On the other hand, the monitor could fall back to our default approach if tracing was activated. In contrast, our single-stepping method (Sections 5.3.2 and 5.3.3) is not limited to tracing only Linux kernels and can single-step functions at arbitrary locations by considering corner-cases, which affect the control-flow. This renders our design capable of monitoring all guest kernel locations.

Besides, our single-stepping schemes create up to three `altp2m` views, whereas each view maps an individual variant of the original page (Fig. 5.6). Also, we consume up to two additional pages per page holding the target function. Instead of creating an additional *step-view*, we can use a *backup page* holding the original instruction and a second `SMC` per tap point (Fig. 5.2(a)). Thus, we can reduce the pages as one *backup page* has capacity for up to 512 tap points (on AArch32).

5.5.2 Limitations

We have presented novel approaches for stealthy monitoring the kernel space on ARM, in multi-vCPU guest domains. Our implementation of Xen `altp2m` on AArch64, which we combine with a *de-synchronization* technique of the split-TLBs on AArch32 to counteract the lack of *execute-only* memory, allows us to hide arbitrary code from the guest. Yet, our prototype entails the following limitations.

Applicability: By employing `SMC` instructions as a trigger to switch the control-flow to the VMM, we limit ourselves to only intercepting the execution of EL1, which is the guest's kernel space. Even though, with Xen `altp2m`, we provide the necessary means to also hide arbitrary code in user space, we chose the `SMC` instruction to implement tap points because guests cannot subscribe to `SMC` events (Sec. 5.3.1). This strategy reduces the complexity of the monitor, as it eliminates the need for any `SMC` event injections into the guest. A great alternative to the `SMC` instruction is the Branch Exchange Jazelle (`BXJ`) instruction [BVN16]. This instruction can be executed in `EL0` and can be configured to trap into `EL2`. Yet, while AArch32 guests can execute `BXJ`, AArch64 guests do not support this instruction. Yet, we can implement yet another alternative by injecting instructions, which access dedicated memory that is known to be protected by the monitor.

Robustness: Our *Split-TLB-SS* solution cannot guarantee ongoing stealthy operation on AArch32 CPUs that implement a uniTLB. On such systems, attackers can re-synchronize the TLBs and reveal the hidden `SMC` instructions from inside the guest. To achieve this, they must synchronize the TLBs to detect pages hidden in the iTLB. This can be achieved by forcing the system to evict the primed iTLB entries to the uniTLB and fetch them to the dTLB. In this context, the attacker must ensure that the dTLB does not contain valid mappings of the affected pages, as they would satisfy the guest's read and write requests. Also, if the primed entries in the iTLB get evicted, while having valid mappings in the dTLB, the VMM will be able to re-prime the iTLB. As such, the guest must only evict the dTLB. Explicit evictions result from flush operations. Implicit evictions result when the iTLB or dTLB buffer gets full and needs to store a new entry; the hardware evicts one of the entries from either the iTLB or dTLB to the uniTLB. The adversary knows that ARM holds a finite number (in our case 10) of fully associative entries in the iTLB and dTLB. Upon memory allocation, she can read or write to the allocated memory pages forcing the system to implicitly flush the dTLB. Hence, the next data access to the target page will consult the uniTLB. Next, the attacker flushes the iTLB to the uniTLB in a similar way to ensure the primed entries are available for future access through the dTLB. Subsequent reads from the target address will consult the uniTLB and reveal the hidden `SMC`.

Contrary to the ARM architecture, x86 reserves a *global bit* as part of the system's page table entries. The kernel can set the *global bit* in the page table entries to cause the system to avoid flushing the TLB entries that hold their respective translations; the system can still evict these entries, yet, only on a high TLB pressure. Thus, even if ARM implemented a similar bit, it would not be enough to fully cloak the analysis on AArch32.

Scope: On AArch32, we further restrict our scope to tracing the execution of code pages in kernel space, which do not perform integrity checks on themselves. This applies to the majority of the kernel, including system call handlers. This limitation emerges as we cannot hide injected `SMC` instructions at arbitrary positions in the kernel space; AArch32 specifies executable pages to be marked with *read-execute* access privileges. Thus, for instance, we distance our mechanisms from monitoring dynamically loaded kernel modules that might perform integrity checks of routines located on the same page as the checking mechanisms.

Besides, the analyst must be aware of system calls that are mapped into the `vDSO`, a shared library mapped into the address space of user space applications. It is used to increase performance of frequently called system calls, by eliminating the context-switch overhead. This way, frequently called system calls are mapped to user space and therefore cannot be monitored through `SMC` instructions. On ARM, however, symbols that are exported through the `vDSO` into user space are limited (i.e., there exist only four symbols on AArch64 and two symbols on AArch32).

Stealth: Some of the problematic anti-virtualization categories deal with behavioral discrepancies between physical and virtual environments (Sec. 4.4.2). These comprise timing overhead induced by emulation or analysis. Side effects of certain instructions can differ as they are not sufficiently documented. This category can only be partially addressed. The hardware behavioral knowledge can be gathered through massive testing [SAM14] and simulated by a VMM. Yet, if an attacker has access to external time sources, such as Network Time Protocol (NTP), she will be able to detect discrepancies caused by the virtualization overhead.

5.5.3 Malicious Abuse of Stealthy Tracing Primitives

As it is often the case in IT security, we have to be aware that strong and, in particular, stealthy analysis primitives can turn into a serious threat in hands of malicious actors (Sec. 4.4). For instance, adversaries can combine the introduced stealthy analysis primitives with on-demand VMM deployment strategies to introduce powerful rootkits (Chap. 4). The same applies to curious administrators, who could utilize the discussed analysis primitives to violate their customers' privacy by dynamically monitoring VMs [HB17, MHHW18, WMA+19, MPR+21]. Without any additional means, the customers do not have the necessary foundation to trust their infrastructures completely. In order to be able to establish trust into otherwise untrusted environments, we envision that future research and cloud providers will continue investing into modern advances of confidential computing architectures [Int21, Adv20, Arm21].

5.6 Related Work

This chapter focuses on stealthy monitoring on ARM, which applies architectural quirks to cloak analysis artifacts from the guest. In the following, we present previous research upon which we build the foundation for stealthy analysis on ARM.

VMM-based analysis: An analysis framework must be stealthy to avoid perturbing malware. SPIDER [DZX13] is a stealthy debugging and instrumentation framework based on Linux KVM. By leveraging the EPT mechanism, SPIDER splits selected code and data pages of the guest's physical memory to implements invisible breakpoints. Similarly to our work, this strategy essentially creates two different mappings of the particular guest physical page. By exchanging the page mappings in the global set of EPT tables, SPIDER mediates access to the page that holds the breakpoint by switching between the *read-only* (data) and the *execute-only* (code) view. However, contrary to our work, using a single set of EPTs per VM can induce race conditions when hiding software breakpoints in multi-*vCPU* environments, a problem-scenario we discuss in-depth in Sec. 5.3.1. In addition, DRAKVUF [LMP+14] is a VMI-based, automated dynamic malware analysis framework built on top of LibVMI [Lib20] and Xen. DRAKVUF extends the idea of SPIDER by automating the process of malware analysis, which is not limited to the kernel but can be also applied to user space applications. Similar to our work, DRAKVUF is capable of tracing multi-vCPU environments by leveraging Xen `altp2m` [Len16]. Through our work, we extend DRAKVUF to support stealthy monitoring on ARM (Sec. 5.4.2). Finally, the WhiteRabbit [PKZ18] VMI framework combines on-the-fly virtualization with VMI on x86 and ARM (Chap. 4). Our primitives combined with WhiteRabbit would establish a stealthy monitor that could be deployed on systems not explicitly set up for VMI.

Beyond VMM-based analysis: SPECTRE [ZLSS13] facilitates a stealthy analysis framework by operating in the SMM on x86, a level below the hypervisor. In a similar fashion, MALT [ZLS+15] provides debugging capabilities that can be employed from remote. Even though SMM-based introspection approaches dissociate themselves from VMI, they essentially employ similar techniques. SPROBES [GVJ14] utilizes ARM TrustZone to enforce kernel integrity. Specifically, Ge et al. instrument SPROBES in form of `SMC` instructions at critical locations in the OS kernel that could be abused by adversaries. Contrary to our work, `SMC` instructions cause the system to trap into a handler in the secure world. Similarly, Ninja [NZ17] leverages TrustZone and also ARM's performance monitor unit to transparently analyze malware. Also, Lengyel et al. [LKPE14, LKE15] explore concepts that may be leveraged for VMI with Xen on ARM.

TLB de-synchronization and maintenance: The Shadow Walker rootkit [SB05], abuses the split-TLBs for stealth purposes. Similarly, Wurster et al. [WvOS05] defeat integrity checks. Additionally, the MoRE Shadow Walker [Tor14] demonstrates that modern, hybrid TLB organizations with an additional shared TLB level are prone to de-synchronization techniques. Grsecurity, on the other hand, de-synchronizes the split-TLB architecture in `PAGEEXEC` [PaX20] to overcome the lack of hardware supported *execute-only* pages. Finally,

Wang et al. [WWZ+19] present a novel technique that allows to intercept TLB misses on x86. To achieve this, the authors set the *reserved bits* in the page table entries. As a result, every TLB miss that consults the page tables the affected pages results in a page fault. These mechanisms mainly focus on the x86 architecture. Also, the presented approaches require invasive kernel changes or a dedicated hypervisor. In contrast, our approach employs capabilities of the open source Xen hypervisor to de-synchronize TLBs on both ARMv7 and ARMv8 architectures facilitating stealthy monitoring of guest domains.

5.7 Summary

In this chapter, we have turned our attention towards the objective Q1. In particular, we have investigated whether the modern ARM architecture is equipped with the necessary hardware support that is required by stealthy VMI. In this regard, we have identified that both AArch32 and AArch64 lack the foundation required to set and single-step breakpoints inside VMs in a stealthy way. Even though the modern ARM architecture is not equipped with the necessary hardware capabilities, we have proposed novel techniques that facilitate stealthy monitoring of guest OSes on ARM. We overcame ARM's lack of hardware support for stealthy VMI by presenting alternative primitives which empower stealthy monitoring of guest OSes. These primitives establish an alternate way of setting and single-stepping software breakpoints without using the intended hardware mechanisms. We have extended the Xen Project hypervisor to leverage SLAT to define and dynamically switch among different guest-physical memory views. To this end, we have introduced the first system on ARM that is capable of holding multiple guest memory views in parallel. Combined with alternative methods for placing and single-stepping breakpoints, this capability presents a stealthy solution on AArch64. We have further combined the above techniques with peculiarities of the TLB organization to overcome the lack of *execute-only* memory on AArch32. Specifically, we have de-synchronized the TLB organization on ARM to hide software breakpoints in the guest's memory. Finally, we have equipped the dynamic binary analysis framework, DRAKVUF, with the alternative monitoring primitives and used it to examine and identify the inherent advantages and limitations of our techniques. In conclusion, to answer the objective Q1, we believe that our methodologies can establish powerful covert VMI analysis systems on ARM.

As we have shown in this chapter, by combining software techniques with the system's hardware virtualization capabilities in a sophisticated way, we can overcome many of the hardware's limits and establish powerful techniques that were not originally foreseen. Equipped with this knowledge, in the next chapter, as part of the objective Q2, we begin investigating novel in-guest primitives, which can enhance the security of OS components.

Chapter 6

Selective Memory Protection

If we knew what we were doing, it wouldn't be called research, would it?
— ALBERT EINSTEIN

We have established that hardware virtualization extensions present an effective and powerful asset for security analysts (Chap. 4 and 5). The capabilities of virtualization extensions, repurposed for VMI, open security experts new ways that allow them to improve their investigation. In this chapter, we shift our research focus away from VMI towards virtualization-assisted OS security. Specifically, we change our perspective of an isolated, stealthy, and analysis-oriented point of view, towards novel concepts for OS architectures, which leverage virtualization extensions for defense purposes. We envision an OS architecture that alleviates the strict separation between the OS and a VMM; virtualization extensions can be utilized by the OS kernel directly, without the need for a fully-fledged VMM. In other words, the OS kernel can either (i) dedicate a subsystem or (ii) retrospectively deploy virtualization-assisted services by installing a thin VMM, such as WhiteRabbit (Chap. 4), with the sole purpose of taking control of the system's virtualization extensions. In both cases, it becomes the task of the OS kernel to define flexible security policies, without having to export the logic to the VMM. Consequently, we investigate how to integrate virtualization extensions into the OS kernel to enhance selected OS components with the ability to defend against the implications of memory corruption vulnerabilities (Q2).

Attackers leverage memory corruption vulnerabilities to establish primitives for *reading* from or *writing* to the address space of a vulnerable process. These primitives form the foundation for code-reuse and data-oriented attacks. While various defenses against the former class of attacks have proven effective, mitigation of the latter remains an open problem. During the past three decades, data-oriented attacks have evolved from a theoretical

exercise [YM87] to a serious threat [CXS+05, Syn14, CBP+15, HSA+16, SXC17, IAJP18]. During the same time, we have witnessed a plethora of effective security mechanisms that prompted attackers to investigate new directions and exploit less-explored corners of victim systems. Specifically, recent advances in Control-Flow Integrity (CFI) [ABEL05, BHK+14, CZM+14, MBBM15, WBD+16], Code Pointer Integrity (CPI) [KSP+14, ZH18], and code diversification [PaX03, CLH+15, BHR+15] have significantly raised the bar for code-reuse attacks. In fact, CFI schemes have been adopted by Microsoft [Mic20b], Google [Goo20b], and LLVM [LLV20].

Code-reuse attacks chain short code sequences, dubbed *gadgets*, to hijack an application's control-flow. It suffices to modify one control-flow structure, such as a function pointer or a return address, with the start of a crafted gadget chain, to cause an application to perform arbitrary computation. In contrast, data-oriented attacks completely avoid changes to the control-flow. Instead, these attacks aim to modify *non-control data* to cause the application to obey the attacker's intentions [HSA+16, SXC17, IAJP18]. Typically, attackers leverage memory corruption vulnerabilities that enable arbitrary *read* or *write primitives* to take control over the application's data. Stitching together a chain of *data-oriented gadgets*, which operate only on data, allows attackers to either disclose sensitive information or escalate privileges, without violating the application's control-flow. This way, data-oriented attacks remain under the radar, despite code-reuse mitigations, and can have disastrous consequences [Syn14]. We anticipate further growth in this direction in the near future, and emphasize the need for practical primitives that eliminate such threats.

Researchers have suggested different strategies to counter data-oriented attacks. Data-Flow Integrity (DFI) [CCH06] schemes dynamically track a program's data flow. Similarly, by introducing memory safety to the C and C++ programming languages, it becomes possible to completely eliminate memory corruption vulnerabilities [NMW02, JMG+02, NZMZ09, NZMZ10]. While both directions have the potential to thwart data-oriented attacks, they lack practicality due to high performance overhead, or suffer from compatibility issues with legacy code. Instead of enforcing data-flow integrity, researchers have started exploring isolation techniques that govern access to sensitive code and data regions [LZC+15, KCB+17, CAGN17]. Still, most approaches are limited to user space, focus on merely protecting a single data structure, or rely on policies enforced by a hypervisor.

In this chapter, we leverage virtualization extensions of Intel CPUs to establish *selective memory protection (xMP) primitives* that have the capability of thwarting data-oriented attacks. Instead of enhancing a hypervisor with the knowledge required to enforce memory isolation, we take advantage of Intel's EPTP switching capability to manage different views on guest-physical memory, from inside a VM, without any interaction with the hypervisor. For this, we extended Xen `altp2m` [LMP+14, PLM+18] and the Linux memory management system to enable the selective protection of sensitive data in user or kernel space by isolating sensitive data in disjoint *xMP domains* that overcome the limited access permissions of the MMU. A strong attacker with arbitrary *read* and *write* primitives cannot access the xMP-protected data without first having to enter the corresponding xMP domain. Furthermore, we equip in-kernel management information and pointers to sensitive data

in xMP domains with authentication codes, whose integrity is bound to a specific context. This allows xMP to protect pointers and hence obstruct data-oriented attacks that target the xMP-protected data. Note that even though we employ Xen to utilize the system's virtualization-extensions, we do not explicitly need a fully-fledged VMM to protect the operating system. Instead, our OS primitives directly utilize the system's virtualization extensions to enhance selected components.

In the following sections, we leverage xMP to protect two sensitive kernel data structures that are vital for the system's security, yet are often disregarded by defense mechanisms: *page tables* and *process credentials*. In addition, we demonstrate the generality of xMP by guarding sensitive data in common, security-critical (user-space) libraries and applications. Lastly, in all cases, we evaluate the performance and effectiveness of our xMP primitives.

Note: Parts of this chapter have been published in [PMG+20]. We have open sourced the code developed as part of this project.[1]

[1]Selective Memory Protection: `https://github.com/virtsec/xmp`

6.1 Memory Partitioning and Isolation Capabilities on Intel

In this section, we discuss Memory Protection Keys [Int20a], Intel's extension to the x86 ISA for fine-grained memory isolation, which can challenge data-oriented attacks in user space. In addition, we provide an overview of the Xen `altp2m` subsystem for the Intel architecture [LMP+14, PLM+18]. Contrary to the introduced Xen `altp2m` subsystem for the ARM architecture in Sec. 5.3.3, we focus on Intel's virtualization extensions, which additionally authorize the guest to switch among different `altp2m` views, without having to explicitly interact with the VMM. This functionality provides the basis for the implementation of our xMP primitives.

6.1.1 Memory Protection Keys

Intel's Memory Protection Keys (MPK) technology supplements the general paging mechanism by further restricting memory permissions. In particular, each paging structure entry dedicates four bits that associate virtual memory pages with one of 16 protection domains. These domains correspond to sets of pages whose access permissions are controlled by the same *protection key* (`PKEY`). User-space processes control the permissions of each `PKEY` through the 32-bit `PKRU` register. Specifically, MPK allows different `PKEY`s to be simultaneously active, and page table entries to be paired with different keys to further restrict access to the associated pages. For each `PKEY`, the thread-local `PKRU` register holds two bits (*write disable* and *access disable*) that define access permissions of the corresponding protection domain. Data accesses to protection domains are thus restricted by both the protection key and page table access permissions. Intel MPK allows threads to individually partition memory that belongs to their address space into 16 (at most) domains, and to constrain access to individual domains without affecting domains of other threads.

A benefit of MPK is that it allows user threads to independently and efficiently harden the permissions of large memory regions. For instance, threads can revoke *write* access from entire domains without entering kernel space, walking and adjusting page tables, and invalidating TLBs; instead, threads can just set the *write disable* bit of the corresponding `PKEY` in the `PKRU` register. Another benefit of MPK is that it extends the access control capabilities of page tables, enabling threads to enforce (i) *execute-only* code pages [CLH+15, PPK+17], and (ii) *non-readable*, yet *present* data pages [CAGN17] by setting the *access disable* bit of the associated `PKEY`. Since the x86 MMU lacks the ability to enforce such policies via page tables, mapped code and data pages can become subject to code-reuse [SMD+13] and data-oriented attacks [Syn14, CXS+05, HCA+15, HSA+16, MWK+18] that result from memory disclosures. These capabilities provide new primitives for thwarting data-oriented attacks, without sacrificing performance and practicality [BHK+14], or resorting to architectural quirks [GEN15] and virtualization [CLH+15, BDOT16, WBD+16].

Although Intel announced MPK in 2015 [Cor15], it was integrated only in 2017, and so far only to the Xeon Skylake-SP family. Later, in 2021, AMD followed Intel and added MPK support to its EPYC Milan CPUs. Unfortunately, both CPU families are dedicated to high-

end servers. Hence, the need for similar isolation features remains on desktop, mobile, and legacy server CPUs. Another issue is that attackers with the ability to arbitrarily corrupt kernel memory can (i) modify the per-thread state (in kernel space) holding the access permissions of protection domains, or (ii) alter protection domain bits in page table entries. This allows adversaries to deactivate restrictions that are enforced by the MMU. Lastly, the isolation capabilities of MPK are geared towards user-space pages. Sensitive data in kernel space thus remains prone to unauthorized access. In fact, at the time of writing, there is no equivalent mechanism for protecting kernel memory from adversaries armed with arbitrary *read* and *write* primitives. Consequently, there is a need for alternative memory protection primitives, the creation of which is the main focus of this work.

6.1.2 The Xen `altp2m` Subsystem

VMMs leverage SLAT (Sec. 2.2.3) to isolate physical memory reserved for VMs [Int20a]. In addition to in-guest page tables that translate guest-virtual to guest-physical addresses, the supplementary SLAT tables translate guest-physical to host-physical memory. Unauthorized accesses to guest-physical memory, which is either not mapped or lacks privileges in the SLAT table, trap into the VMM [Wal02, BDF+03]. As the VMM exclusively maintains the SLAT tables, it can fully control a VM's *view* on its physical memory [DZX13, LMP+14, PKZ18, PLM+18]. Xen's *physical-to-machine* subsystem (`p2m`) [BDF+03, Lin20a] employs SLAT to define the guest's *view* of the physical memory that is perceived by all virtual CPUs (vCPUs). By restricting access to individual page frames, security mechanisms can use `p2m` to enforce memory access policies on the guest's physical memory.

Unfortunately, protecting data through a single *global* view (i) incurs a significant overhead and (ii) is prone to race conditions in multi-vCPU environments. Consider a scenario in which a guest advises the VMM to *read-protect* sensitive data on a specific page. By revoking *read* permissions in the SLAT tables, illegal *read* accesses to the protected page, e.g., due to malicious memory disclosure attempts, would violate the permissions and trap into the VMM. At the same time, for legal guest accesses to the protected page frame, the VMM has to temporarily relax its permissions. Whenever the guest needs to access the sensitive information, it has to instruct the VMM to walk the SLAT tables—an expensive operation. More importantly, temporarily relaxing permissions in the global view creates a *window of opportunity* for other vCPUs to freely access the sensitive data without notifying the VMM.

The Xen *alternate p2m* subsystem (`altp2m`) [LMP+14, PLM+18] addresses the above issues by maintaining and switching between *different* views, instead of using a single, *global view*. As the views can be assigned to each vCPU individually, permissions in one view can be safely relaxed without affecting the active views of other vCPUs. In fact, instead of relaxing permissions by walking the SLAT tables, `altp2m` allows switching to another, less restrictive view. Both external [LMP+14, PLM+18] and internal monitors [LZC+15, SCL+18] use `altp2m` to allocate and switch between views (Chap. 5). Although `altp2m` introduces a powerful means to rapidly change the guest's memory view, it requires hardware support to establish primitives that can be used by guests for isolating selected memory regions.

6.1.3 In-Guest EPT Management

Xen `altp2m` was introduced to add support for the Intel virtualization extension that allows VMs to switch among Extended Page Tables (EPTs), Intel's implementation of SLAT tables [Int20a]. Specifically, Intel introduced the unprivileged `VMFUNC` instruction to enable VMs to switch among EPTs without involving the VMM—although `altp2m` has been implemented for Intel [dml15] and ARM [PLM+18] (Sec. 5.3.3), in-guest switching of `altp2m` views is available to Intel only. Intel uses the VMCS to maintain the host's and the VM's state per vCPU. The VMCS holds an EPT pointer (EPTP) to locate the root of the EPT. In fact, the VMCS has capacity for up to 512 EPTPs, each representing a different view of the guest's physical memory; using `VMFUNC`, a guest can choose among 512 EPTs.

To pick up the above scenario, the guest can instruct the system to isolate and relax permissions to selected memory regions, on-demand, using Xen's `altp2m` EPTP switching. Furthermore, combined with another feature, i.e., the Virtualization Exceptions (`#VE`), Xen `altp2m` allows in-guest agents to take over EPT management tasks. More precisely, the guest can register a dedicated exception handler that is responsible for handling EPT access violations; instead of trapping into the VMM, the guest can intercept EPT violations and try to handle them inside a (guest-resident) `#VE` handler.

6.2 Threat Model

By shifting our focus away from VMI towards virtualization-assisted OS security (Q2), we target attackers, who pursue offensive strategies to subvert the OS kernel. In this chapter, we extend the *offensive* attacker strategy in Sec. 3.2, which facilitates the attacker to mount data-oriented attacks. Yet, to be able to counter these with the proposed defense mechanisms we expect the system to additionally support the state-of-the-art defense mechanisms against code injection and code-reuse attacks.

Specifically, we expect the system to be protected from code injection [KKP03] through Data-Execution Prevention (DEP) or other proper W^X policy enforcement, and to employ Address Space Layout Randomization (ASLR) both in kernel [Lia09, Edg13] and user space [Cor04, PaX03]. Also, we assume that the kernel is protected against return-to-user (ret2usr) [KPK12] attacks through SMEP/SMAP [Yu11, Cor12b, Int20a]. Other hardening features, such as Kernel Page Table Isolation (KPTI) [Cor17, GLS+17], stack smashing protection [vdV06], and toolchain-based hardening [Ope19a], are orthogonal to xMP—we neither require nor preclude the use of such features. Moreover, we anticipate protection against state-of-the-art code-reuse attacks [STL+15, CBP+15, ELO+15, GKK+18] via either (i) fine-grained CFI [BCN+17] (in kernel [GTPJ16] and user space [TRC+14]) coupled with a shadow stack [Cor18], or (ii) fine-grained code diversification [Lar14, KCL+18], and with execute-only memory (available to both kernel [PPK+17] and user space [CLH+15]).

Assuming the above state-of-the-art protections prevent an attacker from gaining arbitrary code execution, we focus on defending against attacks that leak or modify sensitive data in user or kernel memory [CLH+15, PPK+17], by transforming memory corruption vulnerabilities into *arbitrary read* and *write primitives*. Attackers can leverage such primitives to mount data-oriented attacks [MWK+18, IAJP18] that (i) disclose sensitive data, such as cryptographic material, or (ii) modify sensitive data structures, such as page tables or process credentials.

6.3 Selective Memory Protection Primitives

To fulfil the need for a practical mechanism for the protection of sensitive data, we identify the following requirements:

❶ *Partitioning* of sensitive kernel and user-space memory regions into disjoint domains.

❷ *Isolation* of memory domains through fine-grained access control capabilities.

❸ *Context-bound integrity* of pointers to memory domains.

Although the x86 architecture allows for memory partitioning through segmentation or paging ❶, it lacks fine-grained access control capabilities for effective memory isolation ❷ (e.g., there is no notion of *non-readable* pages; only *non-present* pages cannot be read). While previous work isolates user-space memory by leveraging unused, higher-privileged x86 protection rings [LSK18], the isolation of the kernel memory is primarily achieved through Software-Fault Isolation (SFI) solutions [PPK+17]. Even though the page fault handler could be extended to interpret selected *non-present* pages as *non-readable*, switching permissions of memory regions that are shared among threads or processes on different CPUs can introduce race conditions: granting access to isolated domains by relaxing permissions inside the *global* page tables may reveal sensitive memory contents to the remaining CPUs. Besides, each permission switch would require walking the page tables, and thus frequent switching between a large number of protected pages would incur a high run-time overhead. Lastly, the modern x86 architecture lacks any support for immutable pointers. Although ARMv8.3 introduced the Pointer Authentication Code (PAC) [Qua17] extension, which equips pointers with authentication codes, there is no similar feature on x86. As such, x86 does not meet requirements ❷ and ❸.

In this work, we fill this gap by introducing *selective memory protection (xMP)* primitives that leverage virtualization to define efficient memory isolation domains—called *xMP domains*—in both kernel and user space, enforce fine-grained memory permissions on selected xMP domains, and protect the integrity of pointers to those domains (Fig. 6.1). In the following, we introduce our xMP primitives and show how they can be used to build practical and effective defenses against data-oriented attacks in both user and kernel space. We base xMP on top of x86 and Xen [Lin20a], as it relies on virtualization extensions that are exclusive to the Intel architecture and are already used by Xen. Still, xMP is by no means limited to Xen, as we further discuss in Sec. 6.7.1. Furthermore, xMP is both backwards compatible with, and transparent to, non-protected and legacy applications.

6.3.1 Memory Partitioning through xMP Domains

To achieve meaningful protection, applications may require multiple *disjoint* memory domains that cannot be accessible at the same time. For instance, an xMP domain that holds the kernel's hardware encryption key must not be accessible upon entering an xMP

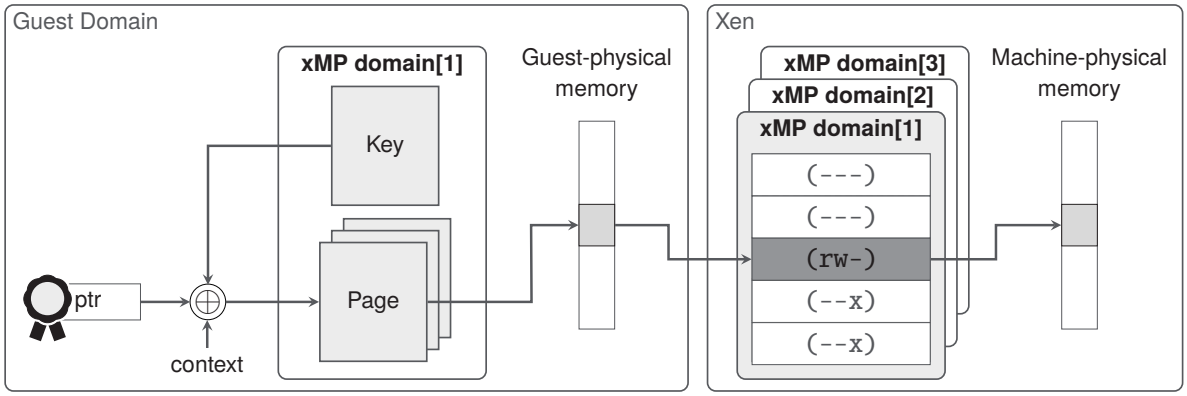


Figure 6.1: xMP uses different Xen `altp2m` views, each mapping guest frames to machine frames with different access permissions, to partition memory into isolated xMP domains. In addition, equipping data pointers to protected memory with HMACs establishes context-bound pointer integrity.

domain containing the private key of a user-space application. The same applies to multi-threaded applications in which each thread maintains its own session key that must not be accessible by other threads; otherwise a potentially compromised thread could access their session keys. We employ Xen `altp2m` as a building block for providing disjoint xMP domains (Sec. 6.1.2). An xMP domain may exist in one of two states, the permissions of which are configured as desired. In the *protected state*, the most restrictive permissions are enforced to prevent data leakage or unauthorized modification. In the *relaxed state*, the permissions are temporarily loosened to enable legitimate access to the protected data.

The straightforward way of associating an `altp2m` view with each xMP domain is not feasible because only a single `altp2m` view can be active at a given time. Instead, to enforce the access restrictions of all xMP domains in each `altp2m` view, we propagate the permissions of each domain across all available `altp2m` views. Setting up an xMP domain requires at least two `altp2m` views. Regardless of the number of xMP domains, we dedicate one view, the *restricted view*, to unify the memory access restrictions of all xMP domains. We configure this view as the default on every vCPU, as it collectively enforces the restrictions of all xMP domains. We use the second view to *relax* the restrictions of (i.e., *unprotect*) a given xMP domain and to allow legitimate access to its data. We refer to this view as *domain[id]*, with *id* referring to the xMP domain of this view. By entering *domain[id]*, the system switches to the `altp2m` view *id* to bring the xMP domain into its relaxed state—crucially, all other xMP domains remain in their protected state. By switching to the *restricted view*, the system switches *all* domains to their protected state.

To accommodate n xMP domains, we define $n + 1$ `altp2m` views. Fig. 6.2 illustrates a multi-domain environment with *domain[n]* as the currently active domain (the page frames of each domain are denoted by the darkest shade). The permissions of *domain[n]* in its relaxed and protected states are `r-x` and `--x`, respectively. The `--x` permissions of

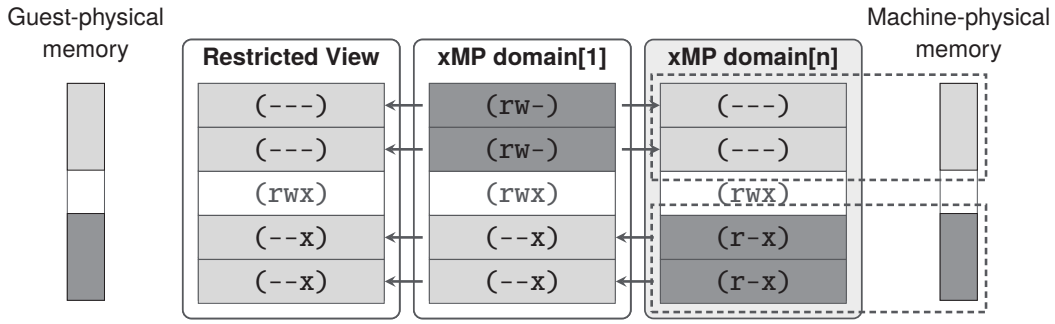


Figure 6.2: The system configures $n + 1$ `altp2m` views to create n disjoint xMP domains. Each $\{domain[i] \mid i \in \{1, ..., n\}\}$ relaxes the permissions of a given memory region (dark shade) and restricts access to memory regions belonging to other xMP domains (light shade). In this example the xMP domain n is the active domain.

domain[n]'s protected state are enforced not only by the restricted view, but also by *all other* xMP domains ($\{domain[j] \mid \forall j \in \{1, ..., n\} \wedge j \neq n\}$). This allows us to partition the guest's physical memory into multiple domains and to impose fine-grained memory restrictions on each of them, satisfying ❶.

An alternative approach to using `altp2m` would be to rely on Intel MPK (Sec. 6.1.1). Although MPK meets requirements ❶ and ❷, unfortunately it is applicable only to user-space applications and cannot withstand abuse by memory corruption vulnerabilities targeting the kernel. Furthermore, even when focusing only on user-space protection using MPK, attackers can still disclose sensitive data in multi-threaded environments. Given that threads share the same page tables, a controlled memory corruption vulnerability in one malicious thread can gain access to protected data as soon as another benign thread relaxes the permissions of an MPK domain. Consequently, due to the limited capabilities of MPK, and since Intel has only recently started shipping (at the time of writing) server CPUs with MPK, we opted for a solution that works on both recent and legacy systems, and can protect both user-space and kernel-space memory.

6.3.2 Isolation of xMP Domains

We establish a *memory isolation primitive* that empowers guests to enforce fine-grained permissions on the guest's page frames. To achieve this, we extended the Xen interface to allow utilizing `altp2m` from inside guest VMs. Specifically, we implemented an API for the Linux kernel that allows the use of existing hypercalls (`HVMOP`s) [dml15] that interact with the Xen `altp2m` subsystem for triggering the VMM to configure page frames with requested access permissions on behalf of the VM. Also, for performance reasons, we extended Xen with a new hypercall: `HVMOP_altp2m_isolate_xmp`. This hypercall enables guests to place selected page frames into xMP domains according to Sec. 6.3.1. Although this hypercall is not vital for xMP (we can substitute its functionality with a set of existing hypercalls to Xen

`altp2m`), it reduces the number of interactions with the hypervisor. Finally, we use `altp2m` in combination with Intel's in-guest *EPTP switching* and the `#VE` feature to allow in-guest agents to take over several EPT management tasks (Sec. 6.1.3). This setup minimizes VMM interventions and thus improves performance. Consequently, we do not have to outsource logic to the VMM or to an external monitor, as the scheme provides flexibility for defining new memory access policies from inside the guest.

Consider an in-guest application that handles sensitive data, such as passwords, cookies, or cryptographic keys. To protect this data, the application can use the *memory partitioning primitives* that leverage `altp2m` to allocate an xMP domain, e.g, *domain[1]* in Fig. 6.2: in addition to *domain[1]* holding original access permissions to the guest's physical memory, our *memory isolation primitive* removes *read* and *write* permissions from the page frame in the *restricted view* (and remaining *domains*). This way, unauthorized *read* and *write* attempts outside *domain[1]* will violate the restricted access permissions. Instead of trapping into the VMM, any illegal access traps into the in-guest `#VE`-handler, which generates a segmentation fault. Upon legal accesses, instead of instructing the VMM to walk the EPTs to relax permissions, the guest executes the `VMFUNC` instruction to switch to the less-restrictive *domain[1]* and serve the request. As soon as the application completes its request, it will use `VMFUNC` to switch back to the *restricted view* and continue execution.

This scheme combines the best of both worlds: flexibility in defining policies, and fine-grained permissions that are not available to the traditional x86 MMU. Our primitives allow in-guest applications to revoke *read* and *write* permissions on data pages, without making them *non-present*, and to configure code pages as *execute-only*, hence satisfying requirement ❷.

6.3.3 Context-bound Pointer Integrity

For complete protection, we have to ensure the integrity of pointers to sensitive data within xMP domains. Otherwise, by exploiting a memory corruption vulnerability, adversaries could redirect pointers to (i) injected, attacker-controlled objects outside the protected domain, or (ii) existing, high-privileged objects inside the xMP domain.

As x86 lacks support for pointer integrity (in contrast to ARM, for which PAC [Qua17, LNW+19] was recently introduced), we protect pointers to objects in xMP domains in software. We leverage the Linux kernel implementation of SipHash [AB12] to compute Keyed-Hash Message Authentication Codes (HMACs), which we use to authenticate selected pointers. SipHash is a cryptographically strong family of pseudorandom functions. Contrary to other secure hash functions (including the SHA family), SipHash is optimized for short inputs, such as pointers, and thus achieves higher performance. To reduce the probability of collisions, SipHash uses a 128-bit secret key to generate HMACs. The security of SipHash is limited by its key and output size. Yet, with pointer integrity, the attacker has only one chance to guess the correct value; otherwise, the application (or the entire system) will crash and the key will be re-generated.

To ensure that pointers cannot be illegally redirected to existing objects, we bind pointers to a specific context that is *unique* and *immutable*. The `task_struct` data structure holds thread context information and is unique to each thread on the system. As such, we can bind pointers to sensitive, task-specific data located in an xMP domain to the address of the given thread's `task_struct` instance.

Modern x86 processors use a configurable number of page table levels that define the size of virtual addresses. On a system with four levels of page tables, addresses occupy the first 48 least-significant bits. The remaining 16 bits are sign-extended with a value dependent on the privilege level: they are filled with ones in kernel space and with zeros in user space [KPK14]. This allows us to reuse the unused, sign-extended part of virtual addresses and to truncate the resulting HMAC to 15 bits. At the same time, we can use the most-significant bit 63 of a canonical address to determine its affiliation—if bit 63 is set, the pointer references kernel memory. This allows us to establish pointer integrity and ensure that pointers can be used only in the right context ❸.

Contrary to ARM PAC, instead of storing keys in registers, we maintain one SipHash key per xMP domain in memory. After generating a key for a given xMP domain, we grant the page holding the key *read-only* access permissions inside the particular domain (all other domains cannot access this page). In addition, we configure Xen `altp2m` so that every xMP domain maps the same guest-physical address to a different machine-physical address. Every time the guest kernel enters an xMP domain, it will use the key that is dedicated to this domain (Fig. 6.1). In fact, by reserving one specific memory page for keys, via the kernel's linker script, we allow the kernel to embed key addresses as immediate instruction operands that cannot be controlled by adversaries (i.e., code regions are immutable). Alternatively, to exclude the compiler from managing key material, we can leverage the VMM to establish a trusted path for generating the secret key and provisioning its location to the VM, e.g., through relocation information [KCL+18]. In particular, by leveraging the relocation entries related to the kernel image, the VMM can replace placeholders at given kernel instructions with the virtual addresses of (secret) key locations, upon the first access to the respective key. Alternatively, we can provide the hypervisor with all placeholder locations through an offline channel [LZC+15]. Regardless of the exact strategy for generating, embedding, and assigning a secret key to a specific xMP domain, once deployed, the keys remain immutable to the guest and can be read-accessed only after having entered the respective xMP domain; software executing outside a sensitive domain cannot access its key.

6.4 Integrating xMP into Linux

We have extended the Linux memory management system to establish memory isolation capabilities that allow us to partition ❶ selected memory regions into isolated ❷ xMP domains. During the system boot process, once the kernel has parsed the `e820` *memory map* provided by BIOS/UEFI to lay down a representation of the entire physical memory, it abandons its early memory allocators and hands over control to its core components. These consist of: (i) the (zoned) buddy allocator, which manages physical memory; (ii) the slab allocator, which allocates physically-contiguous memory in the `physmap` region of the kernel space [KPK14], and is typically accessed via `kmalloc`; and (iii) the `vmalloc` allocator, which returns memory in a separate region of kernel space, i.e., the `vmalloc` arena [PPK+17], which can be virtually-contiguous but physically-scattered. Both `kmalloc` and `vmalloc` use the buddy allocator to acquire physical memory.

Note that (i) is responsible for managing (contiguous) pages frames, (ii) manages memory in sub-page granularity, and (iii) supports only page-multiple allocations. To provide maximum flexibility, we extend both (i) and (ii) to selectively shift allocated pages into dedicated xMP domains (Fig. 6.3); (iii) is transparently supported by handling (i). This essentially allows us to isolate either arbitrary pages or entire slab caches. By additionally generating context-bound authentication codes for pointers referencing objects residing in the isolated memory, we meet all requirements ❶ – ❸.

6.4.1 Buddy Allocator

The Linux memory allocators use *get-free-page* (`GFP_*`) flags to indicate the conditions, the location in memory (zone), and the way the allocation will be handled [BC05a]. For instance, `GFP_KERNEL`, which is used for most in-kernel allocations, is a collection of fine-granularity flags that indicate the default settings for kernel allocations. To instruct the buddy allocator to allocate a number of pages and to place the allocation into a specific xMP domain, we extend the allocation flags. That is, we can inform the allocator by encoding an xMP domain index into the system's `GFP` allocation flags. This allows us to assign an arbitrary number of pages in different memory zones with fine-granularity memory access permissions. Also, through the encoded xMP domain index the allocator receives sufficient information to inform the Xen `altp2m` subsystem to place the allocation into a particular xMP domain (Fig. 6.3). We use 8 free bits in the allocation flags to encode the domain index, effectively supporting up to 256 distinct domains—more domains can be supported by redefining `gfp_t` accordingly. This way, we can grant exclusive access permissions to all pages assigned to the target xMP domain, while, at the same time, we can selectively withdraw access permissions to the allocated page from all other domains (Sec. 6.3.1). As such, accesses to pages inside the target domain become valid only after switching to the associated guest memory view managed by Xen `altp2m`. When we assign allocated pages to xMP domains, we record the `PG_xmp` flag into the `flags` field of `struct page`, thereby allowing the buddy allocator to reclaim xMP-protected pages at a later point.

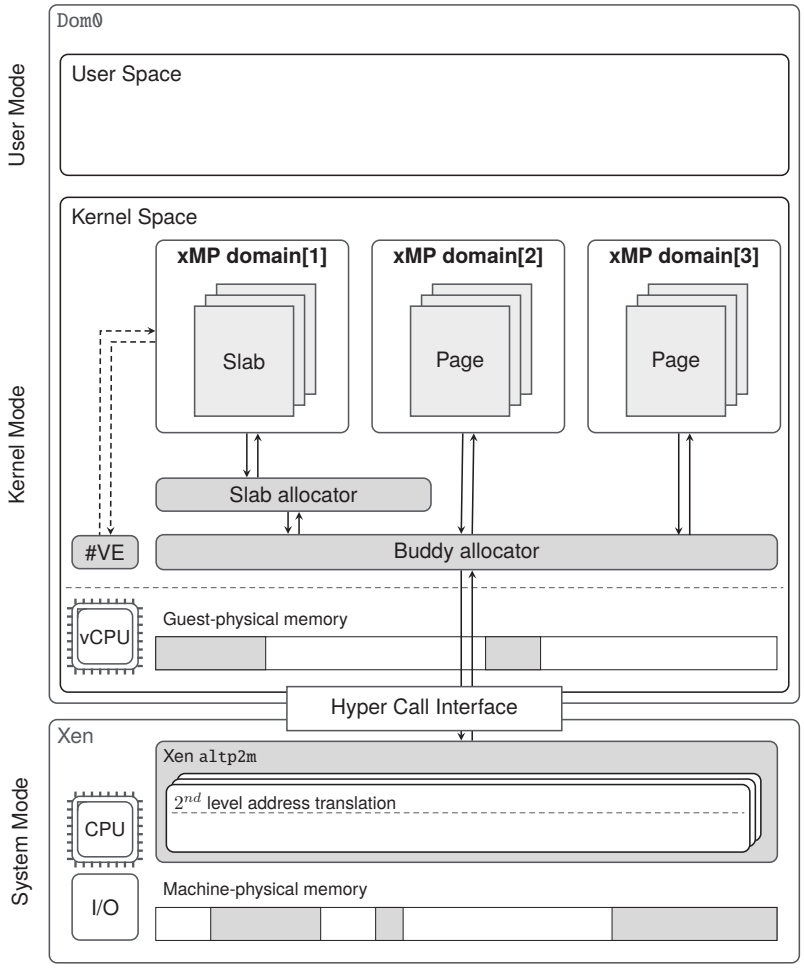


Figure 6.3: Extensions to the Linux *slab* and *buddy allocator* facilitate shifting allocated pages and slabs into xMP domains enforced by Xen `altp2m`.

6.4.2 Slab Allocator

The slab allocator builds on top of the buddy allocator to subdivide allocated pages into small, sub-page sized objects (Fig. 6.3). This scheme allows to reduce internal fragmentation that would otherwise be introduced by the buddy allocator. More precisely, the slab allocator maintains *slab caches* that are dedicated to frequently used kernel objects of the same size [Bon94]. For instance, the kernel uses a cache, named `task_struct`, to maintain all instances of the `struct task_struct` data structure. (Note that the names of the slab caches do not necessarily have to correspond to the names of the data structures.) Such caches allow the kernel to allocate and free objects in a very efficient way, without the need for explicitly retrieving and releasing memory for every kernel object allocation. Historically, the Linux kernel has used three slab allocator implementations: `SLOB`, `SLAB`, and `SLUB`, with the latter being the default slab allocator in modern Linux kernels.

Every slab cache groups collections of continuous pages into so-called *slabs*, which are sliced into small-sized objects. Disregarding further slab architecture details, as the allocator manages slabs in dedicated pages, this design allows us to place selected slabs into isolated xMP domains using the underlying buddy allocator. To achieve this, we extend the slab implementation so that we can provide the xMP domain index in the `GFP` flags, when we create a new slab cache. Consequently, every time the slab cache requests further pages for its slabs, it causes the buddy allocator to shift the allocated memory into the specified xMP domain (Sec. 6.4.1). Note that we can apply this scheme either to all or to selected slabs in the system.

6.4.3 Switches across Execution Contexts

The Linux kernel is a preemptive, highly-parallel system that must preserve the process-specific or thread-specific state on (i) *context switches* and (ii) *interrupts*. To endure context switches, and also prevent other threads from accessing isolated memory, it is essential to include the index of the thread's (open) xMP domain into its persistent state.[2]

6.4.3.1 Context Switches

In general, operating systems associate processes or threads with a dedicated data structure, the Process Control Block (PCB): a container for the thread's state that is saved and restored upon every context switch. On Linux, the PCB is represented by the `struct task_struct`. We extended `struct task_struct` with an additional field, namely `xmp_index_kernel`, representing the xMP domain the thread resides in at any point in time. We dedicate this field to store the state of the xMP domain used in kernel space. By default, this field is initialized with the index of the *restricted view* that accumulates the restrictions enforced by every defined xMP domain (Sec. 6.3.1). The thread updates its `xmp_index_kernel` only when it enters or exits an xMP domain. This way, the kernel can safely interrupt the thread, preserve its open xMP domain, and schedule a different thread. In fact, we extended the scheduler so that on every context switch it switches to the saved xMP domain of the thread that is to be scheduled next. To counter switching to a potentially corrupted `xmp_index_kernel`, we bind this index to the address of the `task_struct` instance in which it resides. This allows us to verify the integrity and context of the index before entering the xMP domain ❸ (Sec. 6.3.3). Since adversaries cannot create valid authentication codes without knowing the respective secret key, they will neither be able to forge the authentication code of the index, nor reuse an existing code that is bound to a different `task_struct`.

[2]Threads in user space enter the kernel to handle system calls and (a)synchronous interrupts. Specifically, upon interrupts, the kernel reuses the `task_struct` of the interrupted thread, which must be handled with care.

6.4.3.2 Hardware Interrupts

Interrupts can pause a thread's execution at arbitrary points. In our prototype, accesses to memory belonging to any of the xMP domains are restricted in interrupt (IRQ) context. (We leave the primitives for selective memory protection in IRQ contexts for future investigation.) To achieve this, we extend the prologue of every interrupt handler and cause it to switch to the *restricted view*. This way, we prevent potentially vulnerable interrupt handlers from illegally accessing protected memory. Once the kernel returns control to the interrupted thread, it will cause a memory access violation when accessing the isolated memory. Yet, instead of trapping into the VMM, the thread will trap into the in-guest `#VE` handler (Sec. 6.1.3). The `#VE` handler, much like a page fault handler, verifies the thread's eligibility and context-bound integrity by authenticating the HMAC of its `xmp_index_kernel`. If the thread's eligibility and the index's integrity is given, the handler enters the corresponding xMP domain and continues the thread's execution. Otherwise, it causes a segmentation fault and terminates the thread.

6.4.3.3 Software Interrupts

The above extensions introduce a restriction with regard to *nested* xMP domains. Without maintaining the state of nested domains, we require every thread to close its active domain before opening another one; by nesting xMP domains, the state of the active domain will be overwritten and lost. Although we can address this requirement for threads in *process context*, it becomes an issue in *interrupt context*: the former executes (kernel and user space) threads that are tied to different `task_struct` structures, while the latter interrupts the process context and reuses the `task_struct` of interrupted threads.

Contrary to hardware interrupts that disrupt the system's execution at arbitrary locations, the kernel explicitly schedules software interrupts (`softirq`) [BC05b], e.g., after handling a hardware interrupt or at the end of a system call. As soon as the kernel selects a convenient time slot to schedule a `softirq`, it will temporarily delay the execution of the active process and reuse its context for handling the pending `softirq`. As such, the kernel handles `softirq` events at seemingly arbitrary times and thus adds irregular delays to the execution of the processes. Generally, work outsourced into a `softirq` cannot access the state of a certain thread. Thus, without considerable adjustments of the `softirq` mechanism, there is no way to associate a `softirq` with a specific thread, on behalf of which it is executed. This is because `softirq`s execute in the context of arbitrarily-selected processes.

The Linux kernel configures 10 `softirq` vectors, with one dedicated for the Read-Copy-Update (RCU) mechanism [McK07]. A key feature of RCU is that every update is split into (i) a *removal* and (ii) a *reclamation* phase. While (i) removes references to data structures in parallel to readers, (ii) releases the memory of removed objects. To free the object's memory, a caller registers a callback that is executed by the dedicated `softirq` at a later point it time. If the callback accesses and frees memory inside an xMP domain, it must first enter the associated domain. Yet, as the callback reuses the `task_struct` instance of an arbitrary thread, it must not update the thread's index to its open xMP domain.

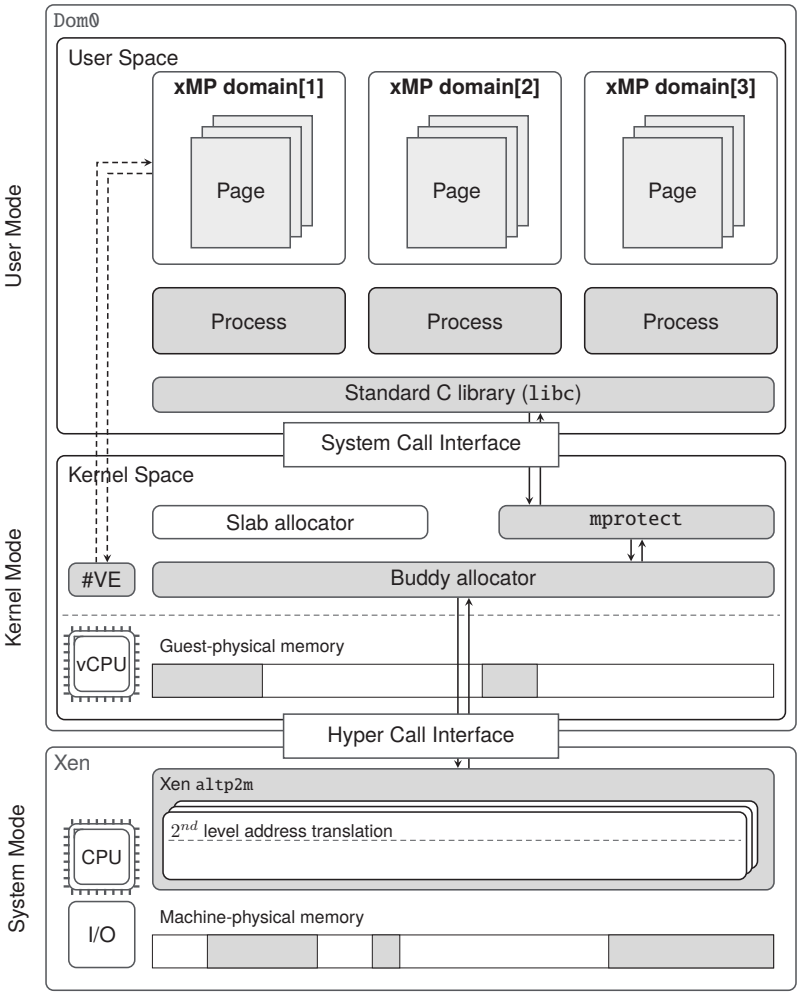


Figure 6.4: User-space applications interact with the Linux kernel through `mprotect` to configure independent xMP domains.

To approach this issue, we leverage the callback-free RCU feature of Linux (by activating the switch `CONFIG_RCU_NOCB_CPU`). Instead of handling RCU callbacks in a `softirq`, the kernel dedicates a thread to handle the work. This simplifies the management of the thread-specific state of open xMP domains, as we can bind it to each task individually: if the thread responsible for executing RCU callbacks needs to enter a specific xMP domain, it can do so without affecting other tasks. As is the case with hardware IRQs, xMP does not allow deferring work that accesses protected memory in `softirq` context.

6.4.4 User Space API

To counter unauthorized manipulation and information disclosure attacks, we grant user processes the ability to protect selected memory regions; we extend the Linux kernel with four new system calls that allow processes to use xMP in user space (Fig. 6.4). Specifically, applications can dynamically allocate and maintain disjoint xMP domains in which sensitive data can remain safe (❶ – ❷). Furthermore, we ensure that attackers cannot illegally influence a process' active xMP domain state by binding its integrity to the thread's context, and hence meet requirement ❸.

Linux has provided an interface for Intel MPK since kernel v4.9. This interface comprises three system calls, `sys_pkey_{alloc,free,mprotect}`, backed by `libc` wrapper functions for allocating, freeing, and assigning user space memory pages to protection keys. Applications use the unprivileged `WRPKU` instruction to further manage memory access permissions of the corresponding protection keys (Sec. 6.1.1). Likewise, we have implemented the system calls `sys_xmp_{alloc,free,mprotect}`, which utilize `altp2m HVMOP`s (Sec. 6.3.2) for allowing programmers to dynamically allocate and maintain different xMP domains in user space. In fact, these system calls implement functionality equivalent to Intel MPK on Linux; they can be used as a transparent alternative on legacy systems without sufficient hardware support (❶ – ❷). On `sys_xmp_mprotect` invocations, we isolate the target virtual memory area (Sec. 6.3.2) and tag it so that we can identify protected memory and release it upon memory reclamation.

Contrary to the MPK implementation of Linux, we do not use the unprivileged `VMFUNC` instruction in user space. Instead, we provide the Linux kernel with an additional system call, namely `sys_xmp_enter`, which enters a requested, previously allocated xMP domain (either more or less restricted) and updates the state of the currently active xMP domain. We save the respective state inside the `xmp_index_user` field of `mm_struct` that is unique to every thread in user space. Also, we bind this index to the address of `mm_struct` (❸). This enables the kernel to verify the integrity and context of the xMP domain index on context switches—in other words, the kernel has the means to detect unauthorized modifications of this field and immediately terminate the application. Note that, with regard to our threat model, we anticipate orthogonal defenses in user space that severely restrain attackers to data-oriented attacks (Sec. 6.2). By further removing `VMFUNC` instructions from user space, and mediating their execution via `sys_xmp_enter`, we avoid unnecessary Return-Oriented Programming (ROP) (or similar code-reuse) gadgets, which could be (ab)used to illegally switch to arbitrary xMP domains.

6.5 Use Cases

We demonstrate the effectiveness and usefulness of xMP by applying it to: (a) *page tables* and *process credentials*, in the Linux kernel; and (b) sensitive *in-process data* in four security-critical applications and libraries.

6.5.1 Protecting Page Tables

With Supervisor Mode Execution Protection (SMEP) [Yu11], the kernel cannot execute code in user space; adversaries have to first inject code into kernel memory to accomplish their goal. Multiple vectors exist that allow attackers to (legitimately) inject code into the kernel. In fact, system calls use the routine `copy_from_user` to copy a user-controlled (and potentially malicious) buffer into kernel memory. While getting code into the kernel is easy, its execution is obstructed by different security mechanisms. For instance, $W \oplus X$ withdraws *execute* permissions from the memory that contains data copied from user space. In addition, defenses based on *information hiding*, such as Kernel Space Address Layout Randomization (KASLR) [Edg13], further obstruct kernel attacks but are known to be imperfect [SMD+13, GLS+17, LSG+18, KHF+19]. Once adversaries locate the injected code, they can abuse memory corruption vulnerabilities, e.g., in device drivers or the kernel itself, to compromise the system's page tables [DGLS17]. This, in turn, opens up the gate for code injection or kernel code manipulation. Consequently, ensuring the integrity of page tables is an essential requirement, which remains unfulfilled by existing kernel hardening techniques [LHKS15, DKD+15, DGLS17].

Our goal is to leverage xMP to prevent adversaries from illegally modifying (i) page table contents and (ii) pointers to page tables. At the same time, xMP has to allow the kernel to update page table structures from authorized locations. With the exception of the initial page tables that are generated during the early kernel boot stage, the kernel uses the buddy allocator to allocate memory for new sets of page tables. Using the buddy allocator, we move every memory page holding a page table structure into a dedicated xMP domain, to which we grant *read-write* access permissions (Sec. 6.4.1), and limit the access of remaining domains to *read-only*. As the kernel allocates the initial page tables statically, we manually inform Xen `altp2m` to place affected guest-physical page frames into the same domain. Every *write* access from outside the dedicated xMP domain results in an access violation that terminates the process. Thus, we must grant access to the protected paging structures to the kernel components responsible for page fault handling and process creation and termination, by enabling them to *temporarily* enter the xMP domain. This scheme does not disrupt the kernel's functionality and complies with requirements ❶ and ❷.

In addition, we extend the kernel's process and thread creation functionality to protect the integrity of every `pgd` pointer referencing the root of a page table hierarchy. More precisely, we equip every `pgd` pointer with an HMAC (Sec. 6.3.3), and verify its integrity every time the pointer gets written to `CR3` (the control register holding the address of the page table root). This protects the pointer against corruptions: as long as adversaries do

not know the secret key, they cannot create a valid HMAC. Attackers cannot read the secret key as it remains inaccessible from outside of the target domain. Attackers also cannot adjust the pointer to the key, as its address is compiled as an *immediate operand* into kernel instructions, and is thus immutable.

On the other hand, we cannot bind the `pgd` pointer to a specific thread context, as kernel threads inherit the `mm_struct` of interrupted user threads. This, however, does not weaken our protection. From the attackers' perspective, it is impossible (or at least very hard) to redirect the `pgd` to a different location, as they do not know the key. One attack scenario is to exchange the `pgd` pointer with a different `pgd` that holds a valid authentication code for another existing thread. Yet, this strategy would not allow the attacker to inject a new address space, but likely crash the application. Note that while we can choose to bind the `pgd` to the address of the associated `mm_struct`, this would not increase its security. As such, we achieve immutability of the page table pointer (❸).

We highlight that even with KPTI [GLS+17, LSG+18] (the Meltdown mitigation feature of Linux that avoids simultaneously mapping user and kernel space), it is possible to authenticate `pgd` pointers. As KPTI employs two consecutive pages, with each mapping the root of page tables to user or kernel space, we always validate both pages by first normalizing the `pgd` to reference the first of the two pages. Lastly, a kernel that leverages xMP to protect page tables does so in a transparent manner to user (and legacy) applications.

6.5.2 Protecting Process Credentials

Linux kernel credentials describe the properties of various objects that allow the kernel to enforce access control and capability management. This makes them an attractive target for data-oriented privilege escalation attacks.

Similarly to protecting paging structures, our goal is to prevent adversaries from (i) illegally overwriting process credentials in `struct cred` or (ii) redirecting the `cred` pointer in `task_struct` to an injected or existing `struct cred` instance with higher privileges. With the exception of reference counts and keyrings, once initialized and committed, process credentials do not change. Besides, a thread may only modify its own credentials and cannot alter the credentials of other threads. These properties establish inherent characteristics for security policies. In fact, LSM [MSKH02] introduce hooks at security-relevant locations that rely upon the aforementioned invariants. For instance, *SELinux* [LS01] and *AppArmor* [GA07] use these hooks to enforce Mandatory Access Control (MAC). Similarly, we combine our kernel memory protection primitives with *LSM* hooks to prevent adversaries from corrupting process credentials.

Linux prepares the slab cache `cred_jar` to maintain `cred` instances. By applying xMP to `cred_jar`, we ensure that adversaries cannot directly overwrite the contents of `cred` instances without first entering its xMP domain (Sec. 6.4.2). As we check both the integrity and context of the active xMP domain index (`xmp_index_kernel`), adversaries cannot manipulate the system to enter an invalid domain (Sec. 6.4.3). At the same time, we allow legitimate *write* access to `struct cred` instances, e.g., to maintain the number of

subscribers; we guard such code sequences with primitives that enter and leave the xMP domain right before and after updating the data structures. Consequently, we meet requirements ❶ and ❷.

As every `cred` instance is uniquely assigned to a specific task, we bind the integrity of every `cred` pointer to the associated `task_struct` at process creation. We check both the integrity and the assigned context to the `task_struct` inside relevant LSM hooks. This ensures that every interaction related to access control between user and kernel space via system calls is granted access only to non-modified process credentials. Consequently, we eliminate unauthorized updates to `cred` instances without affecting normal operation (❸). Again, a kernel that uses xMP to harden process credentials does so in a completely transparent way to existing applications.

6.5.3 Protecting Sensitive Process Data

An important factor for the deployment of security mechanisms is their applicability and generality. To highlight this property, we apply xMP to guard sensitive data in OpenSSL under Nginx, `ssh-agent`, mbed TLS, and `libsodium`. In each case, we minimally adjust the original memory allocation of the sensitive data to place them in individual pages, which are then assigned to xMP domains. Specifically, using the system calls introduced in Sec. 6.4.4, we grant *read-write* access to the xMP domain holding the sensitive data pages, to which remaining domains do not have any access. We further adjust authorized parts of the applications to enter the domain just before *reading* or *writing* the isolated data—any other access from outside the xMP domain crashes the application. In the following, we summarize the slight changes we made to the four applications for protecting sensitive data. Note that an xMP-enabled kernel is backwards compatible with applications that do not make use of our isolation primitives.

OpenSSL (Nginx): The popular *OpenSSL* library offers cryptographic services, including encrypted transmission of data via Secure Socket Layer (SSL) and Transport Layer Security (TLS). Web servers employ OpenSSL to establish secure communication channels. Our adjustments to OpenSSL move private keys into xMP domains. In more detail, OpenSSL uses the `BIGNUM` data structure to manage prime numbers [Ope19b]. We add macros that allocate these structures into a separate xMP domain. Instrumenting the widely-used library OpenSSL allows to protect a wide range of applications. In our case, combining the modified OpenSSL with the Nginx web server (in HTTPS mode) offers protection against memory disclosure attacks, such as Heartbleed [Syn14].

`ssh-agent`: To avoid repeatedly entering passphrases for encrypted private keys, users can use `ssh-agent` to keep private keys in memory, and use them for authentication when needed. This makes `ssh-agent` a target of memory disclosure attacks, aiming to steal the stored private keys. To prevent this, we modify the functions `sshbuf_get_(c)string` to safely store unencrypted keys in dedicated xMP domains.

mbed TLS: The mbed TLS library manages prime numbers and coefficients of type `mbedtls_mpi` in the `mbedtls_rsa_context` [ARM19b]. We define the new data structure `secure_mbedtls_mpi` and use it for the fields `D`, `P`, `Q`, `DP`, `DQ`, and `QP` in the data structure `mbedtls_rsa_context`. We further adjust the `secure_mbedtls_mpi` initialization wrapper to isolate the prime numbers in an exclusive domain.

`libsodium` (`minisign`): The minimalistic and flexible `libsodium` library provides basic cryptographic services. By only adjusting the allocation functionality of the library, in `sodium_malloc` [Lib19], we enable tools such as `minisign` to safely store sensitive information in xMP domains.

6.6 Evaluation

To evaluate our work, we have implemented the introduced selective memory protection primitives for Linux. In line with the presented use cases in Sec. 6.5, we have applied the primitives to critical kernel and user space data structures to assess the added performance penalty and security.

6.6.1 System Setup

Our setup consists of an unprivileged domain `DomU` running the Linux kernel v4.18 on top of the Xen hypervisor v4.12. In addition, we adjusted the Xen `altp2m` subsystem so that it is used from inside guest VMs, as described in Sec. 6.3 and 6.4. The host is equipped with an 8-core 3.6GHz Intel Skylake Core i7-7700 CPU, and 2GB of RAM available to `DomU`. Although we hardened the unprivileged domain `DomU`, the setup is not specific to unprivileged domains and can be equally applied to privileged domains, such as `Dom0`.

6.6.2 Performance Evaluation

Performance is a critical aspect of modern OSes. New exploit mitigation technologies are unlikely to be employed in practice if they incur a significant run-time overhead. To evaluate the performance impact of xMP we conducted two rounds of experiments, focusing on the overhead incurred by protecting sensitive data in kernel and user space. All reported results correspond to vanilla Linux vs. xMP-enabled Linux (both running as `DomU` VMs), and are means over 10 runs. Note that the virtualization overhead of Xen is negligible [PKZ18] (Sec. 4.3.2) and is therefore disregarded in our setting.

6.6.2.1 Kernel Memory Isolation

We measured the performance impact of xMP when applied to protect the kernel's *page tables* (PT) and *process credentials* (Cred) (Sec. 6.5.1 and Sec. 6.5.2). We used a set of micro (LMbench v3.0) and macro (Phoronix v8.6.0) benchmarks to stress different system components, and measured the overhead of protecting (i) each data structure individually, and (ii) both data structures at the same time (which requires two disjoint xMP domains).

Table 6.1 summarizes the LMbench results, focusing on *latency* and *bandwidth* overhead. This allows us to get some insight on the performance cost at the system software level. Overall, the overhead is low in most cases for both protected page tables and process credentials. When protecting page tables, we notice that the performance impact is directly related to functionality that requires explicit access to page tables, with outliers related to page faults and process creation (`fork()`). In both cases, the system undergoes a set of operations that open and subsequently close the particular protection domain, every time it accesses the page table structures. Contrary to page tables, we observe that although the kernel accesses the `struct cred` xMP domain when creating new processes, the overhead

Table 6.1: Performance overhead of xMP domains for page tables, process credentials, and both, measured using LMbench v3.0.

	Benchmark	PT	Cred	PT+Cred
Latency	`syscall()`	0.42%	0.64%	0.64%
	`open()/close()`	1.52%	75.74%	78.93%
	`read()/write()`	0.52%	150.84%	149.27%
	`select()` (10 fds)	2.94%	3.83%	3.83%
	`select()` (100 fds)	0.01%	0.31%	0.30%
	`stat()`	-1.22%	52.10%	53.33%
	`fstat()`	0.00%	107.69%	107.69%
	`fork()+execve()`	250.04%	9.36%	259.59%
	`fork()+exit()`	461.20%	7.78%	437.31%
	`fork()+/bin/sh`	236.75%	8.49%	240.64%
	`sigaction()`	10.00%	3.30%	10.00%
	Signal delivery	0.00%	2.12%	2.12%
	Protection fault	1.33%	-4.53%	-1.15%
	Page fault	216.21%	-2.58%	216.56%
	Pipe I/O	17.50%	32.87%	73.47%
	UNIX socket I/O	1.16%	1.45%	2.25%
	TCP socket I/O	10.23%	20.71%	37.13%
	UDP socket I/O	13.42%	21.98%	41.48%
Bandwidth	Pipe I/O	7.39%	7.09%	17.49%
	UNIX socket I/O	0.10%	6.61%	13.40%
	TCP socket I/O	6.89%	5.83%	14.53%
	`mmap()` I/O	1.22%	-0.53%	0.83%
	File I/O	0.00%	2.78%	2.78%

is insignificant. On the other hand, the xMP domain guarding process credentials is heavily used during file operations, which require access to `struct cred` for access control. The impact of the two xMP domains behaves additively in the combined setup (PT+Cred).

To investigate the cause of the performance drop for the outliers (UNIX socket I/O, `fstat()`, and `read()/write()`), we used the eBPF tracing tools [Fle17]. We applied the `funccount` and `funclatency` tools while executing the outlier test cases to determine the hotspots causing the performance drop by extracting the exact number and latency of kernel function invocations. We confirmed that, in contrast to benchmarks with a lower overhead, the outliers call the instrumented LSM hooks [MSKH02] more frequently. In particular, the function `apparmor_file_permission` [GA07] is invoked by every file-related system call. (This function is related to AppArmor, which is enabled in our `DomU` kernel.) In this function, even before verifying file permissions, we validate the context-bound integrity of a given pointer to the process' credentials. Although this check is not limited to this function, it is performed by every system call in those benchmarks and dominates the number of calls to all other instrumented kernel functions. For every pointer authentication, this function triggers the xMP domain to access the secret key required to authenticate the respective pointer. To measure the time required for this recurring

Table 6.2: Performance overhead of xMP domains for page tables, process credentials, and both, measured using Phoronix v8.6.0.

	Benchmark	PT	Cred	PT+Cred
Stress Tests	AIO-Stress	0.15%	5.87%	5.99%
	Dbench	0.43%	4.74%	3.45%
	IOzone (R)	-4.64%	26.9%	24.2%
	IOzone (W)	0.82%	4.43%	7.71%
	PostMark	0.00%	7.52%	7.52%
	Thr. I/O (Rand. R)	2.92%	7.58%	10.13%
	Thr. I/O (Rand. W)	-5.35%	3.01%	-1.29%
	Thr. I/O (R)	-1.06%	19.54%	20.08%
	Thr. I/O (W)	1.34%	-1.61%	-0.27%
Applications	Apache	6.59%	9.33%	11.14%
	FFmpeg	0.14%	0.43%	0.00%
	GnuPG	-0.66%	-1.31%	-2.13%
	Kernel build	11.54%	1.84%	12.71%
	Kernel extract	2.89%	3.65%	5.91%
	OpenSSL	-0.33%	-0.66%	0.99%
	PostgreSQL	4.12%	0.32%	4.43%
	SQLite	1.10%	-0.93%	-0.57%
	7-Zip	-0.30%	0.26%	0.08%

sequence, we used the `funclatency` (eBPF) tool. The added overhead of this sequence ranges between 0.5–1 *µsec*. An additional 0.5–4 *µsec* is required for entering the active xMP domain on every context switch—including switches between user and kernel space on system calls. Consequently, the context-bound integrity checks affect the performance of light-weight system calls, e.g., `read()` or `write()`, in a more evident way than system calls with higher execution times or even without any file access checks. Having identified the hotspot locations, we can focus on optimizing the performance in the future.

Table 6.2 presents the results for the set of Phoronix macro-benchmarks used by the Linux kernel developers to track performance regressions. The respective benchmarks are split into *stress tests*, targeting one specific system component, and *real-world applications*. Overall, with only a few exceptions, the results show that xMP incurs low performance overhead, especially for page table protection. Specifically, we observe a striking difference between the *read* (R) and *write* (W) Threaded I/O tests: while the `pwrite()` system call is hardly affected by xMP, there is a noticeable performance drop for `pread()`. Using the eBPF tracing tools, we found that the reason for this difference is that the default benchmark settings *synchronize* `pwrite()` operations. By passing the `O_SYNC` flag to the `open()` system call, `pwrite()` returns only after the data has been written to disk. Thus, compared to `pread()`, which completes after 1–2 *µsec*, `pwrite()` requires 2–8 *msec*, and the added overhead of `apparmor_file_permission` accumulates and does not affect `pwrite()` as much as it affects `pread()`.

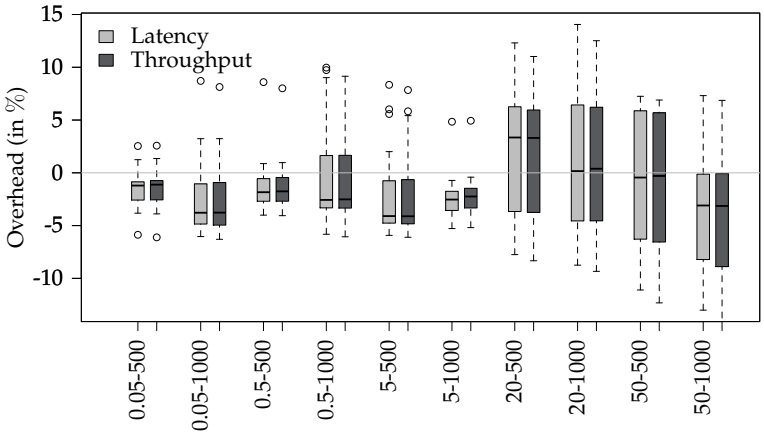


Figure 6.5: Performance impact of xMP on Nginx with varying file sizes and number of connections (X-axis: [file size (KB)]-[# requests]).

6.6.2.2 In-Process Memory Isolation

We evaluated the overhead of in-process memory isolation using our xMP-protected versions of the Nginx and mbed TLS servers (Sec. 6.5.3). In both cases, we used the server benchmarking tool `ab` [Apa19] to simulate 20 clients, each sending 500 and 1,000 requests. To compare our results with related work, we run the Nginx benchmarks with the same configuration used by SeCage [LZC+15]. The throughput and latency overhead is negligible in most cases (Fig. 6.5). Contrary to SeCage, which incurs up to 40% overhead for connections without KeepAlive headers and additional TLS establishment, xMP does not suffer from similar issues in such challenging configurations, even with small files. The average overhead for latency and throughput is 0.5%. For mbed TLS, we used the `ssl_server` example [ARM19a] to execute an SSL server hosting a 50-byte file. (We chose a small file to not mask the overhead with I/O.) On average, the overhead is 0.42% for latency and 1.14% for throughput.

6.6.3 Scalability of xMP Domains

Hardware-based memory isolation features, similar to xMP, support only a small number of domains. For instance, Intel MPK and ARM Domain Access Control (DAC) implement only 16 domains. Nevertheless, we investigate scenarios in which a high number of domains becomes necessary. Modern infrastructures massively deploy OS-level virtualization (i.e., containers), for which Linux namespaces [Ker13b] provide an essential building block by establishing different views on selected global system resources. By integrating xMP into Linux namespaces to isolate selected system resources (Sec. 6.5), we establish (i) the foundation for virtualization-assisted OS-level virtualization, and (ii) the means to evaluate the scalability of xMP domains.

We introduce *xMP namespaces* to isolate process page tables. (Note that xMP namespaces

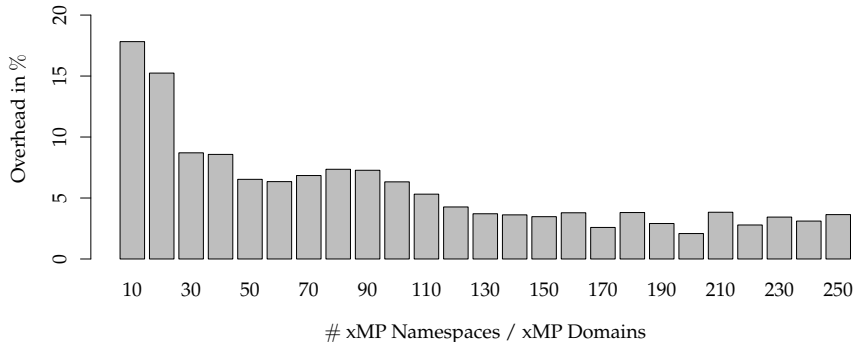


Figure 6.6: Performance impact of up to 250 xMP domains on the scheduler, measured using the customized Phoronix Hackbench stress test.

can be extended to isolate arbitrary data structures.) We use the `unshare()` system call to move a process into a new xMP namespace, effectively placing the process' page tables into a new xMP domain; the process' descendants inherit their parent's xMP namespace, and hence protect their page tables as well. Page tables of processes belonging to different namespaces are isolated by different xMP domains. In this way, we prevent compromised containers from modifying page tables of containers in different xMP namespaces.

To measure the impact of an increasing number of xMP domains, we customized the Phoronix *Hackbench* scheduler stress test. Our adjustments cause the benchmark to place groups of 10 processes each (five senders and five receivers exchanging 50K messages) into separate xMP namespaces. In its standard configuration, Xen supports up to 10 `altp2m` views, with only eight of them being used for xMP domains—`altp2m[0]` is a mirror of the hosts's original view, and must not be changed, and `altp2m[1]` is the restricted view (Sec. 6.3.1). We extended Xen so that we can create up to 256 `altp2m` views; recall that this limitation stems from the fact that we encode the xMP domain's index into the `GFP` allocation flags using eight unused bits (Sec. 6.4.1).

We compare the overhead of an xMP-capable with a vanilla Linux kernel. Fig. 6.6 shows the scheduling overhead of up to 250 distinct xMP namespaces. (Results are means over 10 runs.) Overall, the isolation overhead accumulates linearly with the number of xMP domains—each domain contains the page tables of 10 processes. By increasing the number of processes (250 xMP domains correspond to 2.5K processes), the time required to schedule and run each stress test (i.e., 10 processes exchanging 50K messages) amortizes the overhead, which can even drop to about 2%. This experiment presents the ability of our prototype to scale up to 250 distinct isolation domains, an order of magnitude more than what can be achieved by existing schemes, like Intel MPK and ARM DAC (16 domains).

Lastly, note that in the experiment above, page tables are assigned to isolated xMP domains during process creation, but are populated while the benchmark is executing, due to copy-on-write and dynamic memory allocations. Therefore, the experiment also captures the management overhead of our prototype when it dynamically propagates changes to the corresponding restricted domain views.

6.6.4 Security Evaluation

We evaluated the security of our memory protection primitives using real-world exploits against (i) page tables, (ii) process credentials, and (iii) sensitive data in user space. Despite a strong attacker with arbitrary *read* and *write* primitives to kernel and user memory, by meeting the requirements ❶ – ❸, our system blocks illegal accesses to sensitive data.

6.6.4.1 Attacking the Kernel

We assume an attacker who aims to elevate their privilege using an arbitrary *read* and *write* primitive in kernel memory. To evaluate this scenario, we used a combination of real-world exploits that achieve the aforementioned capability. We first reconstructed an exploit to bypass KASLR [DGLS17]. The `task_struct` of the first process (`init_task`) has a fixed offset to the kernel's base address and is linked to all processes on the system. This provided us with the ability to locate sensitive management information about individual processes, including the root of the page table hierarchy and process credentials. We then abused CVE-2017-16995 (i.e., a sign-extension vulnerability in BPF) to gain an arbitrary read-write primitive to kernel memory.

In the next step, we implemented two different attacks that target (i) the page tables and (ii) the credentials of a given process, respectively. In the first attack, we used the write primitive to modify individual *page table entries* of the target process. This allowed us to grant the *write* permission to (an otherwise *execute-only* mapped) kernel code page with a rarely used system call handler, which is overwritten with shellcode that disables SMEP and SMAP in the `CR4` register. This lends the attacker the power to inject arbitrary code and data into kernel memory. In the second attack, we exchanged the `cred` pointer in the malicious process' `task_struct` with a pointer to an existing `struct cred` instance with higher privileges. In both attacks, we were able to elevate the privileges of the malicious process. By applying xMP to protect page tables and process credentials (Sec. 6.5.1 and 6.5.2), we were able to successfully block both attack scenarios.

To systematically evaluate xMP, we consider attacks that can be equally applied to all kernel structures. We generalize the attack vectors against sensitive kernel structures in the following strategies. Under our threat model, attackers can:

- directly modify the data structure(s) of interest;
- redirect a pointer of a data structure to an injected, attacker-controlled instance;
- redirect a pointer of a data structure to an existing instance with higher privileges.

xMP withstands modification attempts of the protected data structures (❶ – ❷), as only authorized kernel code can enter the associated xMP domains. For instance, when protecting *page tables*, without first hijacking the kernel's execution, the attacker reaches an impasse on how to modify page tables isolated in xMP domains. Injecting code is thus prevented in the first place. Alternatively, the attacker can modify a thread's *pointer* to

a sensitive data structure. In this case, the modified value must comply with the added context-bound integrity (❸) that is enforced on every context-switch or right before accessing the sensitive data structure (Sec. 6.3.3). Since attackers do not know the secret key, they cannot compute an HMAC that would validate the pointer's integrity. Consequently, attackers cannot redirect the pointer to an *injected* data structure.

To sidestep the secret key, attackers could overwrite the pointer with an *existing* pointer (holding a valid HMAC) to a data structure instance with higher privileges. Yet, as pointers to xMP-protected data are bound to the thread's context (❸), attackers cannot redirect pointers to instances belonging to other threads. Note that attackers would have to overwrite the `pgd` pointer of a *privileged* thread with the `pgd` pointer of an *attacker-controlled* thread, when targeting page tables.

6.6.4.2 Attacking User Applications

We chose Heartbleed [Syn14] as a representative data leakage attack due to its high impact. As a result of the lack of a bounds check of the attacker-controlled `payload_length` field of OpenSSL's `HeartbeatMessage`, the attacker can reveal up to 64KB of linear memory that may hold private keys, passwords, and other sensitive information, without altering the application's control flow. By equipping the vulnerable OpenSSL library with the ability to guard secret material (Sec. 6.5.3), we prevented the sensitive regions from leaking. Illegal accesses caused an EPT violation that trapped into the `#VE` handler, which reported the illegal access and terminated the application.

6.6.4.3 Attacking Protection Primitives

Our user-space API does not use the `VMFUNC` instruction, but instead relies on a new system call (Sec. 6.4.4). Given that `VMFUNC` is an *unprivileged* instruction, an attacker can still use it in an attempt to enter different xMP domains. Even if an attacker introduced a `VMFUNC` instruction in the application's memory to mount a `VMFUNC` faking attack [LZC+15], the next context switch would restore the xMP domain's state from `xmp_index_[kernel|user]`, making the kernel immune to illegal domain switches from user space. The attacker could try to use a *write* primitive to modify the kernel's xMP domain state in `xmp_index_[kernel|user]`, forcing the kernel to enter a privileged domain and grant access to sensitive data on the next context switch. Yet, as we bind the integrity of the active xMP domain state to the associated thread's context, any attempt to tamper with it will crash the process.

Further, mediating the execution of `VMFUNC` instructions through the `sys_xmp_enter` system call introduces gadgets that allow switching to previously-allocated xMP domains. Nevertheless, to perform such attacks, the attacker will need to change the application's control flow, something that we assume to be thwarted by orthogonal defenses (Sec. 6.2).

6.6.4.4 I/O Attacks

Compromised I/O devices or drivers can access memory that holds sensitive data. To address this threat, the VMM should confine device-accessible memory (i) by employing the system's IOMMU (e.g., Intel VT-d [Int19]) or (ii) by means of SLAT. The former strategy ensures that sensitive memory in one of the xMP domains will not be mapped by the translation tables of the IOMMU; sensitive data structures become inaccessible to devices. In the latter approach, without IOMMU, the guest is likely to use bounce buffers (e.g., in combination with Virtio [Rus08]) or directly access the devices. In both cases, a corrupted device or driver would access guest-virtual addresses, which are regulated by Xen's `altp2m` subsystem. Thus, it becomes impossible to leak or modify protected information, without first having to gain arbitrary code execution capabilities in kernel mode.

6.7 Discussion

With xMP, we enhance Linux OSes with the capability to thwart data-oriented attacks. In this section we review extensions and alternative application scenarios for xMP, and discuss the limitations of our implementation.

6.7.1 Extensions to xMP

Intel Sub-Page Write Permission: Intel has announced the *Sub-Page Write-Permission (SPP)* feature for EPTs [Int20a] to enforce memory *write* protection on sub-page granularity. Specifically, with SPP, Intel extends the EPT with an additional set of SPP tables that determine whether a 128-byte *sub-page* can be accessed. Selected 4KB guest page frames with restricted *write* permissions in the EPT can be configured to subsequently walk the SPP table to determine whether or not the accessed 128-byte block can be written.

Once this feature is implemented in hardware, it will enrich xMP in terms of performance and granularity. Let us consider the use case of protecting process credentials. Once initialized, the credentials themselves become immutable. However, meta information, such as reference counters, must be updated throughout the lifetime of the `cred` instance. This requires to first enter the xMP domain and relax the permissions to the otherwise *read-only* credentials, before updating the metadata. Using SPP, we can arrange `struct cred` so that all metadata is placed into writable sub-pages, despite the memory access restrictions of the xMP domain.

Execute-Only Memory: A corollary of the lack of *non-readable* memory (Sec. 6.1.1) is that the x86 MMU does not support *execute-only* memory—code pages have to be *readable* as well. This has allowed adversaries to mount Just-In-Time ROP (JIT-ROP) attacks [SMD+13], which can bypass code randomization defenses. By reading code pages, an attacker can harvest ROP gadgets and construct a suitable payload on the fly. A defense against JIT-ROP attacks is thus to enforce execute-only memory to prevent the gadget harvesting phase [BHK+14, GEN15, CZW+17, PPK+17]. By defining execute-only xMP domains for code pages, xMP can offer similar protection.

Alternative Hypervisors and Architectures: Xen is by no means the only system on which xMP can be integrated. Other hypervisors that implement (or can be extended with [QK19]) similar functionality to Xen's `altp2m` can be equally used. Similarly, xMP does not dependent on Intel CPUs, as it does neither require hardware-supported EPTP switching nor the in-guest `#VE` feature—maintaining and switching among different views can be done in software. This would also relax Intel's restriction with respect to the maximum number of EPTPs, as the number of views would not be bound to hardware capabilities. For instance, at the risk of sacrificing performance, we could port xMP to Xen `altp2m` on ARM [PLM+18], in which `altp2m` does not rely on hardware support (Sec. 5.3.3). On ARM, xMP would also benefit from PAC [Qua17] for implementing context-bound pointer integrity. Alternatively, we could use a thin VMM (e.g., WhiteRabbit) with a flexible deployment strategy to dynamically install the foundation for xMP (Chap. 4).

Hypervisor-managed Linear Address Translation: Intel has recently announced a new addition to its virtualization extensions, coined *hypervisor-managed linear address translation (HLAT)* [Int20b]. HLAT defines another set of paging structures that allow VMMs or trusted, and privileged guest kernels to manage translations of *guest-virtual* to *guest-physical* addresses. So far, this was the task of the OS kernel. (The OS kernel governs page tables, which are responsible for (the guest's) linear address translation.) Yet, by additionally combining the HLAT paging structures with the VMM-controlled EPTs, the system facilitates the VMM to ensure the integrity of the combined (i.e., guest-linear to machine-physical) translations; malicious attempts to tamper with the (in-guest) paging structures can thus be obstructed, without having to mark the page frames that hold the guest's page tables as *read-only* in the EPT tables (Sec. 6.5.1). This addition provides hardware support for the concept behind xMP in regard to protecting the guest's page tables.

The HLAT extension provides another benefit for xMP, which is related to the overall performance. Because xMP restricts the page frames holding the guest's paging structures, by marking them as *read-only*, it can obstruct unauthorized access attempts to the page table contents (Sec. 6.5.1). Unfortunately, management-related write-accesses to the page table contents that originate from the *CPU* similarly violate the permissions of the EPT and trap into the `#VE` handler; every time software (*read* or *write*) accesses a page, the CPU updates the *accessed* and *dirty* flags (`A/D`) in the respective page table entry. Thus, enforcing integrity of the paging structures through EPT tables incurs an additional performance overhead for xMP. This is because xMP has to handle CPU-initiated updates of management information in the page table entries. With HLAT, the EPT table entries receive an additional control bit, *page-write*, which allows the processor to update the `A/D` bits, without violating the permissions of the EPT and unnecessarily trapping into the `#VE` handler.

6.7.2 Limitations

The Linux callback-free RCU feature [Cor12a] relocates the processing of RCU callbacks out of the `softirq` context, into a dedicated thread (Sec. 6.4.3.3). This allows RCU callbacks to enter xMP domains without affecting other threads' xMP domain state, as we currently do not provide selective memory protection in IRQ contexts. We leave addressing this issue (i.e., selective memory protection in asynchronous execution contexts) to future work.

Besides, in our most recent implementation, we manually instruct the kernel when to enter a specific xMP domain. Instead, we could automate this step by instructing the compiler to bind annotated data structures to xMP domains. In addition, the compiler could instrument the kernel code with calls that enter/leave the xMP domain immediately before/after accessing the annotated data structure.

Finally, we do not support nested xMP domains. In fact, we prohibit entering domains, without first closing the active domain; by nesting xMP domains, the state of the opened domain will be overwritten. To address this, the kernel needs to securely keep track of the previously opened xMP domains by maintaining a stack of xMP domain states per thread. Note that this relates to adding xMP support in IRQ contexts.

6.8 Related Work

While the possibility of non-control data (or data-oriented) attacks has been identified before [YM87], Chen et al. [CXS+05] were the first to demonstrate the viability of data-oriented attacks in real-world scenarios, ultimately rendering them as realistic threats. With FlowStitch [HCA+15], Hu et al. introduced a tool that is capable of chaining, or rather stitching together, different data-flows to generate data-oriented attacks on Linux and Windows binaries, despite fine-grained CFI, DEP, and, in some cases, ASLR, in place. Hu et al. [HSA+16] further show that data-oriented attacks are in fact Turing-complete. They introduce Data-Oriented Programming (DOP), a technique for systematically generating data-oriented exploits for arbitrary x86-based programs. Similarly, Carlini et al. [CBP+15] achieve Turing-complete computation by using a technique they refer to as Control-Flow Bending (CFB). In contrast to DOP, CFB is a hybrid approach that relies on the the modification of code pointers. Still, CFB bypasses common CFI mechanisms, by limiting code pointer modifications in a way that the modified control-flows comply with CFI policies. Ispoglou et al. [IAJP18] extend the concept of DOP by introducing a new technique they coin as Block-Oriented Programming (BOP). Their framework automatically locates dispatching basic blocks, in binaries that facilitate the chaining of *block-oriented gadgets*, which are then chained together to mount a successful attack.

On the other hand, researchers have started to respond to data-oriented attacks. For instance, DataShield [CP17] associates annotated data types with security sensitive information. Based on these annotations, DataShield partitions the application's memory into two disjoint regions, and inserts bounds checks that prevent illegal data flows between the sensitive and non-sensitive memory regions. Type-based Data Isolation [MvdKG22] dedicates separate memory arenas for colored objects (i.e., object types) and instruments the compiler to constrain pointers such that they remain within the bounds of the respective arena. Similar to our work, solutions based on virtualization maintain sensitive information in disjoint memory views [KCB+17, LZC+15, HDX+18]. While MemSentry [KCB+17] isolates sensitive data, SeCage [LZC+15] additionally identifies and places sensitive code into a secret compartment. Both frameworks leverage Intel's *EPTP switching* to switch between the secure compartment and the remaining application code. Yet, in contrast to our work, MemSentry and SeCage are limited to user space. Also, SeCage adds complexity by duplicating and modifying code that would normally be shared (e.g., libraries) between the secret and non-secret compartments.

EPTI [HDX+18] implements an alternative to KPTI using memory isolation techniques similar to xMP. PrivWatcher [CAGN17] leverages virtualization to ensure the integrity of process credentials. Contrary to our solution, PrivWatcher creates shadow copies of `struct cred` instances, and places them in a *write*-protected region. PT-Rand [DGLS17] protects page tables using information hiding. Zabrocki introduces LKRG [Zab18], a runtime guard for the Linux kernel, ensuring the integrity of critical kernel components by shadowing selected data structures or matching their hashes in a database. Although, LKRG does not employ virtualization, the author considers to use a VMM for

self-protection. Finally, with PARTS [LNW+19], Liljestrand et al. introduce a compiler instrumentation framework to cope with pointer-reuse attacks via the (recently-introduced) ARMv8.3-A pointer authentication features.

6.9 Summary

In this chapter, we have shifted our research focus away from VMI to novel techniques that utilize virtualization to assist the security of OSes. By closely examining the x86 architecture, we have identified ways that allow us to integrate hardware virtualization extensions into the Linux OS to enhance the security of its subsystems (Q2). This scheme has brought us one step closer to our envisioned design of an OS architecture, in which hardware virtualization support becomes an integral part of the OS. Even more, by following the technological trend, we observe that our research coincides with modern demands on virtualization. For instance, similar to xMP, the independently announced hypervisor-managed linear address translation (HLAT) extensions to the Intel architecture were designed to ensure the integrity of paging structures inside VMs [Int20b]. Another example is given by Microsoft's Kernel Data Protection (KDP) extensions, which—very similar to xMP—introduce new primitives to the Microsoft Windows kernel involving Hyper-V to isolate selected kernel memory regions in order to prevent unauthorized access [Mic20c]. As such, in line with the technological trend, we propose novel defenses against data-oriented attacks. Our system, called xMP, leverages Intel's virtualization extensions to set the ground for selective memory isolation primitives, which facilitate the protection of sensitive data structures in both kernel and user space. xMP extends the Linux memory management system to empower software developers with the ability to shift sensitive data structures into disjoint and isolated domains, despite the limited capabilities of the x86 MMU in regard to memory isolation. Even though, throughout this chapter, we mainly focus on the Intel architecture, we have shown that we can apply xMP to the ARM architecture, for which we have already provided the necessary foundation in the previous chapter (Chap. 5); Xen `altp2m` on ARM does not rely on hardware support and can similarly assist our xMP primitives. We further equip pointers to data in isolated memory with authentication codes to thwart illegal pointer redirection. We demonstrate the effectiveness of our scheme by protecting the page tables and process credentials in the Linux kernel, as well as sensitive data in various user applications. In conclusion, we believe that our results demonstrate that xMP is a powerful and practical solution against data-oriented attacks.

Chapter 7

Enhancing Security of Linux Containers

Writing is nature's way of letting you know how sloppy your thinking is.
— Dick Guindon

Recent advances in OS-level virtualization have successfully propagated this, once rarely used, technology to the masses. Generally, OS-level virtualization techniques leverage services of the underlying OS to establish light-weight isolated execution environments, known as containers (Sec. 2.1.2). Prominent container implementations are Linux Containers (LXC) [Ltd20], BSD jails [Rio20], and Solaris Zones [Ora20]; yet it was Docker [Doc20a] that gained the most popularity. Docker has become the de facto standard for containers in both private and industry sectors for shipping various applications in a convenient and platform-independent way. Unlike system virtualization techniques that implement a virtual hardware interface (i.e., the VM), that offers fully-fledged execution environments for OSes, containers share the same OS kernel with their host and solely abstract the view on global kernel resources. As such, potential kernel exploits originated from inside a container can impair the security of other containers on the same system and even the host itself. Unfortunately, reducing the surface of this attack vector receives insufficient attention. This drives us to investigate this direction to assist OS-level virtualization (Q3).

The *principle of least privilege* mandates each entity to access only those resources that are necessary for its execution. Contrary to this concept, modern OS architectures offer applications a uniform interface, i.e., the *system call interface*, that grants access to an immense number of system calls. The number of system calls available to the application is entirely independent of how many system calls the application requires. For instance, the Linux kernel v5.5 comprises 347 distinct system calls, excluding the number of compatibility system calls. Sadly, a vulnerability in one of those system calls has the potential to open the gate to the underlying kernel [Dat17, Inc17, Dat16], despite modern isolation and security

mechanisms, including Linux namespaces, control groups, capabilities, MAC, KASLR, and Supervisor Mode Execution and Access Prevention (SMEP/SMAP). This threat similarly applies to containers, which share the same OS kernel with their host. In fact, most privilege escalation exploits targeting containers on Linux abuse vulnerable system calls to overcome the isolation enforced by the container and Linux kernel [LLW+18].

One way to mitigate this threat is to remove or filter out unnecessary system calls, which are otherwise freely available to applications and containers. For instance, recent advances in *library debloating* [QPY18, RDDC+17, RNMJ17, AJWK+19, KGP19, WKKWK+20] remove code regions in the process' address space that are irrelevant to the program's execution. While library debloating focuses on reducing ROP (or similar code-reuse) gadgets, it does not eliminate the threat; unavoidably remaining gadgets could allow the attacker to still mount an attack that can abuse a vulnerable system call handler—which might not be used by the program—to subvert the system. Contrary, as a last line of defense, the seccomp (Secure Computing) facility of the Linux kernel provides the necessary means to establish system call filters, which restrain applications and containers to use only whitelisted system calls [Cor09]. However, to this point, there is no easy way to tailor seccomp policies suitable for general-purpose applications and containers automatically.

Researchers have suggested different techniques that employ *static* or *dynamic analysis* to identify the system calls required by the application (or container) under test. Dynamic analysis based approaches [Pro03, WLX+17] employ automatic test generation techniques that trigger different program behaviors, which, in turn, identify different system calls required for the particular functionality. This strategy allows collecting most of the system calls that are similarly accessed in production. Nevertheless, dynamic analysis suffers from *incompleteness*, as the triggered execution traces severely limit the analysis; if a particular system call has not been identified by the dynamic analysis, which bases its results on past execution traces, it does not necessarily mean that it will not be called in production. In fact, we identified a set of system calls that were either completely missed or falsely interpreted by the dynamic analysis based policy generation of previous work [WLX+17]—in both cases, normally authorized system calls were blacklisted. Consequently, due to incomplete information, *false negatives* can interrupt and terminate the execution of sandboxed environments, which presents an intolerable inconvenience in production systems.

Even though existing static analysis frameworks [QPY18, AJWK+19, WKKWK+20, DWKJ+20] apply similar techniques, they either focus on library or container debloating, or strongly rely on additional characteristics of the target binaries. Contrary to the concurring tools, our proposed solution can analyze even stripped and closed-source binaries without restrictions. For instance, Quach et al.'s [QPY18] library debloating technique requires recompilation of the analyzed binaries with *clang*, Nibbler [AJWK+19] cannot work with stripped binaries, and both Egalito [WKKWK+20] and `sysfilter` [DWKJ+20] need position-independent code (PIC) in their input binaries. Finally, Confine [GPBP20] combines static *and* dynamic analysis to extract system calls from containers, yet, mainly focuses on analyzing the standard C library, `libc`. As we discuss in this chapter, Confine requires access to the `libc` source code and omits a precise analysis of the system calls

outside `libc`, which can lead to misleading results. In contrast to all of the mentioned examples, our approach does not depend on any binary metadata and works on unmodified, stripped, closed-source, real-world binaries, and therefore complements their analysis.

When put into a different perspective, the problem of identifying a binary's authorized set of system calls can be reduced to *constant propagation*. Previous research has shown excellent results when applying constant folding and constant propagation to reconstruct Control-Flow Graphs (CFGs) from binaries [BGRT05, KV08, KZV09, KV10, BHV11, FLPV15]. Inspired by the achievements of the previous work, we re-purpose the concepts behind constant propagation to automatically derive legitimate system call numbers from binaries to create tailored seccomp policies.

In this chapter, we present Jesse, a system that leverages static analysis to identify system calls for *arbitrary* non-obfuscated ELF binaries. Note, even though this chapter mainly focuses on enhancing the security of Docker containers, Jesse can be similarly applied to conventional ELF binaries outside of containers; containerized programs are nothing else than conventional program binaries and libraries, packaged in an image that is supported by the respective container runtime (Sec. 2.1.2). Specifically, Jesse aims to complement existing tools [WKKWK+20, GPBP20, DWKJ+20], to bridge the gap to binaries that do not support PIC and lack the necessary binary metadata. Given the provided set of system calls, it can be further processed into strong seccomp policies that establish safe environments inside or outside Docker containers. Since the invocations of system calls do not necessarily reveal the requested system call (a system call invocation is merely the execution of the `syscall` instruction),[1] we implement a *constant propagation* technique based on *abstract interpretation* to derive the contextual semantics required to (safely) approximate the set of system calls a container should be authorized to use [CC77].

To assess our system's capability, we have applied Jesse to ELF binaries that are available for Debian `buster` *stable*, and compared our results against existing frameworks. Specifically, we have picked a representative set of $1,064$ ELF binaries, which correspond to the Debian `buster` *base* image, and *manually* verified Jesse's results; the results have shown that Jesse did not introduce any false negatives and demanded human support in only 20 cases (i.e., 2%). We have further employed Jesse to generate seccomp policies for five of the most prominent Docker containers on Docker Hub. On average, we manage to block 54.7% of all system calls and hence significantly increase the effectiveness of the default seccomp policy for Docker containers, which conservatively blocks around 10.6% completely, and up to 20.4% in combination with restrictions of additional Linux capabilities.

Note: This chapter has not yet been published. Yet, at the point of submission of this dissertation, parts of this chapter have been *accepted* in [GPZ23]. The accepted paper bases upon, yet, extends and fully revises the results of the Bachelor's thesis supervised by the author [Gro18].

[1] While we assume the analyzed binaries use the modern 64-bit *fast system call* instruction on x86, `syscall`, the analysis' concepts are by no means limited to it and can be extended to support different system call instruction variants and architectures.

7.1 Binary Analysis and System Call Filtering

There exist two flavors of program analysis techniques: dynamic and static analysis. Due to their inherent characteristics, both flavors differ in the accuracy of the gathered analysis results, as they might produce *false negatives* or *false positives*. We consider as false negatives, system calls that were falsely missed during the analysis and hence not considered in the set of authorized (i.e., whitelisted) system calls. Execution attempts of these excluded system calls cause the program to crash. Dynamic analysis can produce false negatives as it solely relies on the given execution traces of a program and cannot consider all of its execution paths. On the other hand, we consider as false positives, system calls that were (conservatively) recognized as vital during the analysis and hence were whitelisted, at the risk of never being executed. Static analysis can produce false positives as its assumptions abstract unnecessary details and overapproximate the results. Thus, we resort to a static analysis technique, *abstract interpretation* [CC77], to establish profiles holding whitelisted system calls, which we enforce with the help of the Linux Secure Computing mode [Cor09, Cor12c] for individual binaries.

7.1.1 Linux Secure Computing Mode

The Linux Secure Computing mode, known as *seccomp* [Cor09, Cor12c], is a feature of the Linux kernel that confines processes to a set of black- or whitelisted system calls. Optionally, seccomp allows filtering the system call's arguments to further restrain processes. The most restricted mode, `SECCOMP_SET_MODE_STRICT`, grants only four system calls, namely `read()`, `write()`, `exit()`, and `sigreturn()`; any other system call results in an immediate termination of the process. On the other hand, the mode `SECCOMP_SET_MODE_FILTER` delegates the filtering decision to a user-provided BPF program that can be installed per process [MJ93]; at every system call invocation, the in-kernel BPF bytecode JIT compiler executes the BPF program, with the invoked system call number and arguments as input, to enforce a given policy. Thus, the BPF program can decide whether (i) to execute the system call, (ii) to kill the calling process, or (iii) to return an error.

To improve the security of containers, Docker adopted seccomp profiles and simplified their usage. Instead of requiring the user to provide complex BPF programs, Docker expects a JSON-formatted profile that holds the black- and whitelisted set of system calls. The profile further supports fine-grained policies per system call, e.g., by restricting arguments or by binding the system call to a specific Linux capability (e.g., `CAP_SYS_ADMIN`). Docker (*v18.10.0, dev*) ships a default seccomp profile that unconditionally permits 276 (out of 347 available) system calls [Mob20] on Linux kernel (*v5.5)*, which is about 80.4% of the available system calls [Mob20].[2] By combining seccomp with Linux capabilities, the default seccomp profile may grant up to 34 further system calls. Thus, the default seccomp

[2] Docker's default policy white-lists 315 system calls, out of which 41 either do not exist on x86 or are (redundant) compatibility system calls. Another 2 system calls are granted with argument restrictions.

profile is very coarse-grained (it forbids 10.6% of all available system calls completely and up to 20.4% when combined with Linux capability restrictions), the refinement of which is the main focus of this chapter.

7.1.2 Abstract Interpretation

Abstract interpretation [CC77] is a theory based on static analysis that allows approximating the semantics of programs, making it a useful tool for detecting information leaks [ZC11, CC00, WBL+19]. We leverage abstract interpretation to identify system calls relevant to a program's genuine execution to tailor seccomp policies.

To identify a program's semantic properties, abstract interpretation operates on the program's abstract representation. While the founders of abstract interpretation, Cousot and Cousot [CC77], use *finite flowcharts* to represent programs, we resort to the more commonly used program representation through CFGs. Applied to a CFG, abstract interpretation annotates its edges (i.e., transfers of the control-flow) with symbols representing (possibly infinite) sets of program states. Cousot and Cousot model a program's state as a mapping from variables to values. Once the abstract interpretation terminates, the annotated symbols represent sets of states that can be reached at the associated edges; if a program state is not part of the set of the computed program states at a particular edge in question, the program will never reach this state at that edge, independent of the program's input. Yet, the computed sets may hold states that will never be reached. In most cases, an exact computation of all program properties is not feasible [WZH+11]. Thus, abstract interpretation only provides an *overapproximation* of the exact program semantics.

As it is infeasible to store all possible sets of program states, abstract interpretation operates on a user-provided, predefined set of symbols (S). Thus, it functions on abstract symbols instead of the sets of concrete program states. Cousot and Cousot extend S to a complete lattice L that comprises the following elements to assemble the quintuple $L = (S, \leq, \sqcup, \bot, \top)$. The lattice requires a *partial order* relation ($\leq$) and a *join* operation ($\sqcup$) to work on the set of symbols. Both are necessary, as they simulate the *subset relation* ($\subseteq$) and the *union operation* ($\cup$) for the sets of concrete program states. This means, instead of unifying two sets of concrete program states, the abstract interpretation joins the symbols representing the sets. Similarly, instead of applying the subset relation, it uses the *partial order* relation from the lattice. Finally, to form a complete lattice, L must contain a *top* ($\top$) and a *bottom* ($\bot$) element. These represent the largest and the smallest element ($\forall s \in S : \bot \leq s \leq \top$), respectively, with regard to the *partial order* in the set of symbols S. These elements simulate the empty set and the universe of the sets of concrete program states.

Comparable to *concrete program states* that are lifted to an abstract representation in the form of lattice elements ($s \in S$), abstract interpretation requires lifting *concrete operations* on concrete program states to abstract operations on lattice elements. This is where the *interpretation function* ($Int(e, C)$) comes into play. It has to be tailored to the needs such that it specifies how the basic blocks in the CFG operate on the lattice elements. For example, when we would like to determine whether the values of variables in a given

Algorithm 1: The abstract interpretation algorithm which annotates the edges of the program's CFG with user-provided symbols. The annotations represent the sets of all reachable program states at the particular edge.

Input : A CFG $C = (N, E)$, a complete lattice $L = (S, \leq, \sqcup, \bot, \top)$, and an interpretation function $Int(e, C)$, with $e \in E$

Output: A CFG whose edges are annotated with symbols representing sets that contain all reachable program states at the particular edge

```
1  foreach e ∈ E do
2      annotate e with ⊥;
3  repeat
4      changes:=∅;
5      foreach e ∈ E do
6          oldAnnotation:=get annotation at e;
7          newAnnotation:=Int(e, (N, E));
8          if oldAnnotation ≠ newAnnotation then
9              changes:=changes ∪ {(e, newAnnotation)};
10     foreach (e, newAnnotation) ∈ changes do
11         annotate e with newAnnotation;
12 until changes = ∅;
13 return (N, E);
```

program are even or odd, it is the task of the interpretation function to infer whether the computation results in an even or odd value. Similar to Cousot and Cousot, we assume that the interpretation function receives a specific edge of the CFG as the first argument and the complete CFG (including annotations of the previous rounds) as the second argument. The interpretation function computes the new annotation symbol for the particular edge and returns the symbol to the main abstract interpretation algorithm to update the CFG.

Alg. 1 describes the abstract interpretation which statically analyzes programs by annotating the edges of the program's CFG with program states that can be reached at the particular edge. The algorithm runs in rounds until it reaches a *fixpoint* (i.e., executing another round would not change the annotations). As input, the abstract interpretation receives a CFG C comprising the set of nodes N and the set of edges E, a complete lattice L, and an interpretation function $Int(e, C)$. During initialization, the algorithm annotates all edges in the CFG with $\bot$ (lines 1 to 2). Then, the computation enters the main loop (lines 3 to 12), which is responsible for updating the annotations (lines 10 to 11). Specifically, the main loop calls the interpretation function in line 7 until the annotations of the CFG no longer change (i.e., until the abstract interpretation reaches a fixpoint).

To further clarify how to leverage abstract interpretation, in the following section, we assist the reader in going through a detailed example scenario.

7.1.3 Applying Abstract Interpretation

To demonstrate how to leverage abstract interpretation, let us consider a scenario in which we analyze a part of a function to determine whether or not it will always return even values in the general-purpose register `rcx`. As shown in Fig. 7.1, the example program first sets `rcx` to 10, then enters a loop comprising a single `loop` instruction which subtracts one from `rcx` and continues the loop if `rcx` is not zero after the subtraction, and finally, returns if `rcx` equals zero.[3] In this context, we specify a lattice $\dot{L} = (\dot{S}, \dot{\leq}, \dot{\sqcup}, \dot{\bot}, \dot{\top})$ and an interpretation function $\dot{Int}(e, C)$. The lattice $\dot{L}$ comprises the set of symbols $\dot{S} := \{B, E, O, A\}$. The symbol B (*blank*) represents the empty set of states (i.e., we use B to annotate unreachable edges); the symbols O and E describe that `rcx` can only hold *odd* (O) or *even* (E) values, respectively; and the symbol A (*all*) means that `rcx` can hold any (whether even or odd) natural number. Further, the *bottom* ($\dot{\bot}$) and the *top* ($\dot{\top}$) elements the lattice $\dot{L}$ are represented by B and A, respectively. These abstract symbols overapproximate the concrete set of values. That is, if an edge is annotated with A, it means that `rcx` *can* hold any natural number at the particular edge, but it does not have to.

Further, we define the *partial order* ($\dot{\leq}$) relation and the join ($\dot{\sqcup}$) operation of the lattice $\dot{L}$ as follows. B is the smallest and A is the largest element (E and O are incomparable). That is, with $a, b \in \dot{S}$:

$$a \dot{\leq} b : \Longleftrightarrow \ (a = B) \vee (b = A) \vee (a = b)$$

A join with B (the smallest element) always returns the original element. In all the other cases (i.e., $A \dot{\sqcup} E$, $A \dot{\sqcup} O$, $E \dot{\sqcup} A$, $O \dot{\sqcup} A$, $O \dot{\sqcup} E$, and $E \dot{\sqcup} O$), the result is A:

$$a \dot{\sqcup} b := \begin{cases} a & \text{if } b = B \\ b & \text{if } a = B \\ a & \text{if } a = b \\ A & \text{otherwise} \end{cases}$$

Alg. 2 illustrates the interpretation function $\dot{Int}(e, C)$. This function takes an edge e of the CFG as first argument, the CFG C itself as second argument, and returns a lattice element $s \in \dot{S}$ as annotation for the provided edge e. To compute the annotation, the function first joins the annotations of incoming edges of the basic block from which the edge in the first argument originates ($origin(e)$ returns the basic block from which e originates; $incoming(b)$ returns all edges that target b) in lines 1 to 5.

Assuming the basic block is reachable, the interpretation function extracts the instruction (for simplicity, we assume that the basic block holds only one instruction) in line 8 and abstractly simulates its execution. In case the instruction writes to `rcx`, the function

[3] The x86 instruction `loop` first decrements the value in `rcx` and then checks it for 0. In case `rcx` has reached 0, `loop` continues with the next instruction; otherwise, it jumps to the target operand to continue the loop [Int20a].

Algorithm 2: Interpretation function $\dot{Int}(e, C)$ that determines the new annotation in form of a lattice element for the provided edge e.

Input : Edge e and the annotated control-flow graph C
Output: Lattice element $s \in \dot{S}$

1 $bb := origin(e)$;
2 $pre := B$;
3 **foreach** $e' \in incoming(bb)$ **do**
4 $\quad a :=$annotation at edge e';
5 $\quad pre := pre \dot{\sqcup} a$;
6 **if** $pre = B$ **then**
7 $\quad$ return B;
8 $instr :=$extract instruction in bb;
9 **if** $instr$ *writes to* `rcx` **then**
10 $\quad imm :=$extract immediate in $instr$;
11 $\quad$ return $\begin{cases} E & \text{if } imm \bmod 2 = 0 \\ O & \text{otherwise} \end{cases}$;
12 **else if** $instr =$`loop rel` **then**
13 $\quad rel :=$extract jump target in $instr$;
14 $\quad$ **if** e *targets* rel **then**
15 $\quad\quad$ return $\begin{cases} A & \text{if } pre = A \\ O & \text{if } pre = E \\ E & \text{otherwise} \end{cases}$;
16 $\quad$ **else**
17 $\quad\quad$ return $\begin{cases} B & \text{if } pre = E \\ E & \text{otherwise} \end{cases}$;

interprets the immediate value, imm, and returns E or O, depending on whether imm is even or odd (lines 9 to 11). If the extracted instruction is a branch to the beginning of the loop (`loop` rel), the function distinguishes between two cases, which depend on whether the edge is a fallthrough or a jump to the beginning of the loop. The former case returns B if joining the annotations on the incoming edges results in E. This is because `loop` decrements `rcx` and only terminates the loop if the result is zero. Since decrementing an even number can never be zero, this edge will never be taken. In the latter case, the function returns E; `rcx` must be zero (i.e., even) because `loop` will only fall through if `rcx` is zero after the decrement. In case of a jump, we only have to consider that `rcx` is decremented by the `loop` instruction and that it can flip the value from even to odd, and vice versa. In case of A, this does not matter and we preserve it.

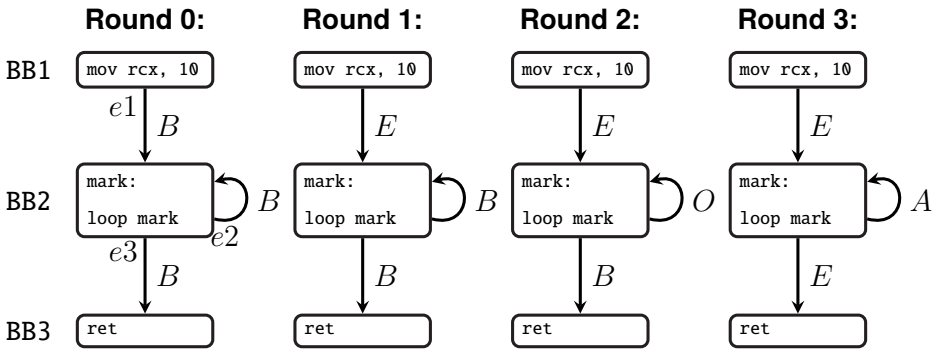


Figure 7.1: We apply the abstract interpretation to the program (in form or a CFG that decrements the value of `rcx` from 10 to 0 in a loop) to determine whether the returned value in `rcx` is *even* (E) or *odd* (O). The iterative annotations of the CFG added by the abstract interpretation are illustrated in four rounds.

To collect the building blocks of the abstract interpretation, Fig. 7.1 illustrates the iterative annotation results of the abstract interpretation applied to our code fragment example in the form of a CFG. We start in `Round 0` (on the left) with all edge annotations set to B. In `Round 1`, the edge leaving `BB1` ($e1$) is annotated with E, because `BB1` always sets `rcx` to 10 which is an even number. The other annotations do not change, because the union of the edges $e1$ and $e2$ in round zero is B meaning that the control-flow will never reach `BB2` in this round. Similarly, the control-flow does not reach the edges $e2$ and $e3$. In `Round 2`, our arguments from `Round 1` apply to $e1$ and the union of $e1$ and $e2$ now returns E. Thus, the edges of $e2$ is annotated with O; decrementing an even value always results in an odd value. The annotation of $e3$ remains B because this edge is not reachable (zero is an even value) and we determined that `rcx` is odd when the control-flow reaches the end of `BB2`. In the last round, we annotate $e2$ with A, because the union of E and O is A and decrementing an arbitrary natural number results in another arbitrary number. In the case of $e3$, the same situation applies initially, but since the edge is only taken when `rcx` contains zero, the analysis recognizes that zero is even and hence annotates $e3$ with E.

7.2 Threat Model

Throughout this chapter, we assume an adversary who pursuits an *offensive* attacker strategy (Sec. 3.2) to subvert and escape benign Docker containers on Linux. Our previous work (Chap. 6) has countered offensive attackers by equipping the OS kernel with strong memory isolation capabilities provided by the hardware virtualization extensions. The extended OS kernel capabilities allowed us, among others, to isolate and guard access to sensitive kernel data structures originated from namespaces of individual containers. Similarly, in this chapter, we protect against adversaries with capabilities to escape Docker containers on Linux. Instead of relying on the system's virtualization extensions, we focus on the available *seccomp* system call filtering capabilities of the Linux kernel, to prevent adversaries, who target a vulnerable system call interface, from taking over the kernel.

In other words, we assume an attacker who intends to escalate her privileges to escape a Docker container by abusing the Linux *system call interface*, the protection of which is the main target of this work. We expect the target Linux kernel to implement the Linux seccomp facility (Sec. 7.1.1). We do not limit the adversarial capabilities to specific attacks, yet, we expect her (i) to exploit memory corruption vulnerabilities to establish *read* and *write* primitives into the victim application's address space in order to hijack its control-flow and (ii) construct code gadgets to exploit kernel vulnerabilities through the system call interface and finally take over the host system. We assume that the containerized application under attack is benign and that the system employs state-of-the-art kernel and user space ASLR [Lia09, Edg13]. We also assume that containerized applications are isolated from other applications via state-of-the-art OS-level virtualization techniques including Linux namespaces [Ker13b], control groups [Bro14], and capabilities [Cor06, Edg15]. Further, we assume the system is protected against code-injection attacks through SMEP/SMAP [Yu11, Cor12b, Int20a] and W^X policy enforcement mechanisms. Defenses against data-oriented attacks [LZC+15, CAGN17, KCB+17, PMG+20] are orthogonal to our work and can be applied independently. The same applies to container [RDDC+17, RNMJ17], configuration [KGP19], and library debloating [QPY18, AJWK+19, KGP19, RDDC+17, RNMJ17] techniques that can further reduce the attack vector.

Under the assumption that the above state-of-the-art defenses restrain the attacker inside the isolated environment, potential memory corruption vulnerabilities in system call handlers can still allow establishing primitives that enable them to read or write to the kernel memory arbitrarily. Such primitives can be abused to (i) leak information to defeat KASLR, (ii) deactivate SMEP/SMAP, and hence (iii) pave the way for the attacker to inject and execute her payload. To limit this attack vector, we concentrate on reducing the system call interface that is available to the attacker by creating seccomp policies tailored for benign Docker containers.

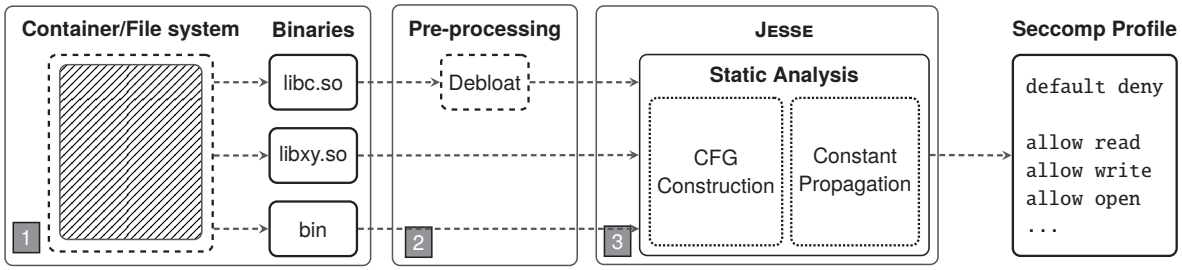


Figure 7.2: Three phases of the seccomp profile generation process. In the first phase, we either extract the binaries to be analyzed from a container image (via container debloating techniques), or directly take them from the file system. In the next phase, we debloat the standard C library *only* and deliver it with the other binaries to JESSE in the third phase to finally generate a seccomp profile.

7.3 System Call Number Analysis

Fig. 7.2 provides a high-level view on the system call number analysis, in which we put JESSE in perspective. The figure partitions the process of deriving seccomp profiles from applications into three main phases. The first phase is responsible for identifying all binaries that are relevant for the analysis. In other words, it is responsible for identifying the complete set of binaries (i.e., programs and shared libraries) on which the target program maintains a dependency. Since the program to be analyzed can be either a binary or a container image, resolving and extracting the dependent binaries can differ; while we can use the ELF header to identify binary dependencies, Sec. 7.4.1 discusses the special case in which we apply container debloating techniques to dissect container images.

The second phase is responsible for identifying all code regions that are associated with each exported function of the *standard C library*. This step has to be done only once and allows JESSE to establish a map that associates each exported library function with a set of system calls. This way, we can refer to the map to identify the system calls that are associated with an exported library function that is used by the target binary in the next phase. We describe this phase in more detail in Sec. 7.3.4.

Finally, the third phase utilizes static analysis to identify the set of system calls a program requires; this phase presents the main focus of this section. We use this set to automatically compile seccomp policies to filter unnecessary and potentially vulnerable system calls, which would have been otherwise freely available. In the following, we present the building blocks that allow us to harvest authorized system calls from ELF binaries. Specifically, we statically analyze programs and their required libraries to locate all `syscall` instructions. Then, we leverage abstract interpretation to identify the system calls. In essence, we divide our analysis into two stages (Fig. 7.2). In the first stage, JESSE assembles a CFG for each function in the given binary and identifies the locations of the `syscall` instructions. The second stage leverages our abstract interpretation scheme to derive the *system call number* that is passed as an argument to the identified `syscall` instruction.

7.3.1 Hypotheses for a Sound Analysis

The design and soundness of Jesse bases on the following four *key hypotheses* (**H1** – **H4**), which we have derived by observing benign, unobfuscated, compiler-generated real-world binaries from official Debian mirrors (we have added the percentage of cases, in which a hypothesis applies in our evaluation (Sec. 7.5), to account for its applicability):

(H1) Most unobfuscated binaries comply with the x86-64 ABI calling conventions (100%) and can be linearly disassembled (99.34%) [ACvdV+16];

(H2) a `syscall` instruction invokes the same system call *in most cases* (98.98%);

(H3) if the binary provides a system call number to a `syscall` instruction by crossing an indirect edge, the edge will not be part of the function holding the `syscall` instruction (100%);

(H4) most binaries do not call system calls directly, instead they utilize wrappers in the standard C library.

(**H1**) presumes a sound and complete disassembly and adherence to the x86-64 ABI calling conventions. Contrary to (**H1**), malicious actors can obstruct disassemblers from correctly interpreting the binary, or build binaries that use arbitrary calling conventions. For instance, by relocating a binary's entry point (even by one byte) on CISC architectures, adversaries can manage to confuse common disassemblers. The same applies to obfuscated binaries or (benign, compiler-generated) code regions that are intertwined with data. Since binary analysis is undecidable [WZH+11], our analysis is inherently limited in its ability to distinguish data from code. Yet, modern binaries strictly separate data from code regions by placing them into disjoint sections. In fact, prior research has shown that even though linear disassembly is an unsolved theoretical problem, most `gcc` and `clang` generated binaries can be linearly disassembled in practice [ACvdV+16]. Thus, we can safely assume that we will be able to recover a sound and complete disassembly from *unobfuscated* and *untampered* binaries.

(**H2**) stems from the fact that distinct system calls are by design inherently different in regard to their behavior and preparations needed to call them; kernel developers prefer adding options to already existing system calls to support new features over introducing new system calls with similar semantics [Cor20a]. Therefore, it is impractical to create functions that are merely responsible for arranging arguments for a set of different system calls. Besides, if such wrapper functions are created nonetheless, they are often small enough to be eligible for inlining, and thus become invisible in the target binary.

(**H3**) states that if we assume that every `syscall` instruction always invokes the same system call handler (**H2**), the binary will hold constant system call numbers, which have either been hard-coded or resolved by the compiler. In fact, we have observed that most `syscall` instructions involve only a few preceding assembly instructions (operating on general-purpose registers) to acquire the associated system call numbers: our experimental

findings confirm that (**H2**) holds in 98.98% of the analyzed system call invocations (Sec. 7.5). Out of the remaining 1.02%, we have observed that none of the system call numbers were provided through an indirect edge *inside the same function* that holds the `syscall` instruction (**H3**). Instead, all of these system call numbers depend on function arguments. As such, Jesse does not consider indirect edges in functions to identify the system call numbers; if Jesse manages to associate a system call number with a `syscall` instruction, all potential indirect calls (inside the function) will invoke the same system call handler (**H2**). (Note that indirect edges delivering system call numbers between functions *not* via function arguments would violate the hypothesis (**H1**)). If such functions have been called indirectly, Jesse cannot be certain about the exact location of the caller that has initiated the function's invocation. In such cases, Jesse requests external expert knowledge (Sec. 7.5.2).

Finally, hypothesis (**H4**) assumes that developers prefer calling system call wrappers in `libc` (such as `fopen()`, `fread()`, and `fwrite()`) over directly invoking system calls. This is because the system call wrappers in `libc` are more convenient and make the resulting program more portable. Hence, if a binary (other than `libc`) invokes a system call directly, it either (i) has highly specific needs regarding the system call's arguments or (ii) the `libc` does not support the desired use case. Consequently, even though binaries (outside `libc`) directly invoke only a few system calls, the system calls they require are of *key-value* for creating seccomp profiles.

Note that every time Jesse cannot deduce a system call number, it will inform and direct the analyst to the exact location of the affected *syscall* instruction to ease further analysis. We discuss the soundness of Jesse in Sec. 7.6.1.

7.3.2 Control-Flow Graph Construction

First, we transform the binary into a CFG, which can be processed by the abstract interpretation (Sec. 7.1.2). A CFG is a tuple (N, E) comprising a set of nodes (i.e., basic blocks) N and a set of edges (i.e., control-flow transfers between basic blocks) E. Since inaccuracies in the CFG can produce incorrect results for the abstract interpretation, it is crucial that the CFG fulfills the following three requirements, whose violation can cause Jesse to misinterpret the target binaries: ① *Correctness of* N; ② *Completeness of* N; and ③ *Correctness of* E. Note that we do not require the set of edges to be complete; we compensate the lack of this property of the CFG through (**H2**) and (**H3**) (refer to Sec. 7.6.1 for details).

The requirement ① ensures the correct contents of basic blocks—it is trivial and met by most CFG construction tools. Yet, not every CFG construction framework assures the requirements ② and ③. Given that binary analysis is undecidable [WZH+11], CFG construction remains an open problem. Often, such frameworks prefer *completeness* over *soundness* of the CFG and thus can falsify the assumptions of analysis frameworks, e.g., through heuristics. For instance, both *angr* [SWH+15, SGS+16] and *radare* [Rad20] can miss basic blocks and/or introduce invalid edges in certain situations. Angr's static analysis based CFG generation tool, CFGFast, can miss code (i.e., violate ②) on indirect branches (i.e., the branch target addresses are computed at run-time) [Ang20]. Respectively, angr's

symbolic execution based CFG generation mechanism, CFGEmulated, can introduce illegal edges (i.e., violate ③), because it relies on symbolic execution to overapproximate the set of possible jump targets [Ang20]. Similar restrictions apply to other CFG reconstruction frameworks [SA.20, BGRT05, KV08, KZV09, KV10, FLPV15]. One of the reasons for applying heuristics is, among others, the tool's generality in application. For instance, heuristics allow to predict the location of code regions to a certain degree (at the risk of potentially violating ② and ③), even for obfuscated binaries or binaries that do not comply with the ABI. Unfortunately, heuristics cannot always provide correct results with certainty, and thus would reduce Jesse's precision. On the other hand, by avoiding heuristics (and by following the hypotheses (**H1** – **H4**)), we can exclude incorrect results with regard to CFG construction with certainty. Generally, any CFG construction tool that follows the aforementioned requirements (① – ③) is an eligible candidate for Jesse. We have decided to apply Lightweight Disassembler (LWD) [Eng20] for our purpose. LWD does not apply any heuristics and its small code base allows us to easily extend and integrate it into Jesse.

Initially, we modify LWD to not regard `syscall` and `call` instructions as control-flow changing instructions. Instead, they are regarded as instructions that can alter registers according to the Linux calling conventions of the x86-64 ABI [MHJM13] (**H1**). In addition, we cause LWD to split the basic blocks holding a `syscall` instruction into two basic blocks that we connect through an artificial edge. This way, we ensure that the second part of the split basic block begins with the `syscall` instruction. This strategy simplifies identifying the system call numbers passed as a parameter for `syscall`.

Contrary to conventional CFG construction frameworks, LWD only considers edges of *direct* branches (including direct relative, absolute, and (un)conditional branches); it avoids heuristics for indirect branches whose targets reside in general-purpose registers (other than `rip`) or memory that cannot be statically determined. For instance, the instructions `ret` and `jmp reg` always conclude a basic block, yet, they do not introduce any edges. Hence, even though the final CFG comprises only correct edges (③), these do not have to be necessarily *complete*. Granted, without considering all code regions, LWD would miss `syscall` instructions. Yet, under (**H1**), this is not the case in our analysis: we modify LWD such that it first linearly disassembles the binary, and then extends the disassembly to a CFG by adding appropriate edges for all direct jumps. This allows us to achieve ① and ②. If we further assume that most `syscall` instructions invoke only *one specific system call* (**H2**), a missing edge (i.e., resulting from incompletely recovered indirect branches) would represent merely an alternative invocation of an already identified system call (Jesse analyses each identified `syscall` instruction). Rarely, a `syscall` instruction is used to call different system calls (Sec. 7.4.2). In such cases, it is impossible for Jesse to determine the system call number statically. As such, Jesse informs the analyst about its exact location (Sec. 7.3.3.3). This setting allows Jesse to either correctly determine the system call number for the respective `syscall` instruction, or, in rare cases, request expert knowledge.

7.3.3 System Call Number Identification

The modern x86-64 architecture implements the *fast system call* instruction, `syscall` [Int20a], that allows user-space applications (`ring 3`) to make use of services provided by the high-privileged kernel (`ring 0`). Upon the execution of the `syscall` instruction, the system switches the context and executes the registered system call dispatcher in kernel space. The system call dispatcher inspects the number passed in the `rax` register to determine which system call handler has been requested [MHJM13]. We apply abstract interpretation to implement an effective constant propagation strategy that allows us to determine the system call number, passed in `rax`, for every identified `syscall` instruction in the CFG of the given binary. In the following, we define a complete lattice $\dot{L}$ and the interpretation function $\dot{Int}(e, C)$ that form our abstract interpretation.

7.3.3.1 Defining the Lattice

The lattice $\dot{L}$ of our abstract interpretation is a quintuple $(\dot{S}, \dot{\leq}, \dot{\sqcup}, \dot{\bot}, \dot{\top})$. The system call numbers passed via `rax` to the system call dispatcher never result from arithmetic computations. Instead, `rax` is either ① initialized with an immutable immediate instruction operand, or ② assigned a value from a different register. While ① can be trivially determined by identifying the assignment of `rax`, pinpointing the exact value in ② must be accomplished through constant propagation techniques. Rarely, system call numbers are read from memory. Even though this property constraints static analysis, Jesse identifies the affected `syscall`'s location and requests the analyst's assistance. To address both cases, we define our lattice to keep track of the constant values that are propagated through the general-purpose registers.[4] Consequently, we define every element $s \in \dot{S}$ to hold either the symbol NX (i.e., the neutral element; the associated edge gets *Never eXecuted*) or a 15-dimensional vector (i.e., the register state). This vector maps each register to an element in the set $\{X, 0, \ldots, 2^{64} - 1\}$ (i.e., to a constant or to an unknown value X). Note that we did not limit our analysis to merely considering the set of existing system call numbers; instead, we decided to consider the complete range of numbers. This decision does not affect the performance and allows us to apply the analysis to system calls that could be potentially added in the future. Formally, $\dot{S}$ adheres to the following definition:

$$\dot{S} := \{NX\} \cup f : R \to \{X, 0, \ldots, 2^{64} - 1\}$$

with R representing the set of general-purpose registers:

$$R := \{\texttt{r[a|b|c|d]x}, \texttt{r[d|s]i}, \texttt{rbp}, \texttt{r[8-15]}\}$$

[4] We exclude `rsp` (stack pointer) and consider only 15 general-purpose registers. These include `rbp` as the compiler can reuse the frame pointer.

Further, we define the *partial order* relation $\dot{\leq}$, with NX as the smallest element. Two vectors $x, y \in \dot{S} \backslash \{NX\}$ fulfill $x \dot{\leq} y$ if they are identical or if elements of y differ only in their mapping to X. We define formally for any $a, b \in \dot{S}$:

$$a \dot{\leq} b : \Longleftrightarrow (a = NX) \vee \bigwedge_{i \in R} ((a(i) = b(i)) \vee (b(i) = X))$$

The join operation $\dot{\sqcup}$ unifies two vectors $a, b \in \dot{S}$. By unifying two vectors, for each element (i.e., register) of the new vector, we preserve the old element's value, if both vector elements are identical. Otherwise, we assign the symbol X to the new element, as the associated register can hold multiple values. Further, if either a or b is assigned the value of the neutral element, NX, the join operation takes the assignment of the respective other lattice element. Thus, the following defines $\dot{\sqcup}$:

$$a \dot{\sqcup} b := \begin{cases} a & \text{if } b = NX \\ b & \text{if } a = NX \\ \lambda r \in R.r \mapsto \begin{cases} a(r) & \text{if } a(r) = b(r) \\ X & \text{if } a(r) \neq b(r) \end{cases} & \text{else} \end{cases}$$

Finally, we specify the bottom element of the lattice $\dot{L}$ ($\dot{\bot}$) to be the smallest one (i.e., $\forall a \in \dot{S} : \dot{\bot} \dot{\leq} a$) and the top element ($\dot{\top}$) to be the largest (i.e., $\forall a \in \dot{S} : a \dot{\leq} \dot{\top}$). Therefore, $\dot{\bot}$ inevitably represents NX. Also, since the vector that assigns all registers to X stands for all possible program states, $\dot{\top}$ represents $\lambda r \in R.r \mapsto X$:

$$\dot{\bot} := NX$$
$$\dot{\top} := \lambda r \in R.r \mapsto X$$

7.3.3.2 Defining the Interpretation Function

Abstract interpretation leverages an interpretation function to project how the basic blocks in the CFG operate on abstract lattice elements that represent a program, or in our case, register states (Sec. 7.1.2). To identify the exact system call number in `rax`, our interpretation function applies symbolic execution [CARB12, SWS+16, SWH+15, SGS+16]. We leverage the symbolic execution framework *angr* [Ang20] to propagate already assigned constant values (in the register state) and to determine the register assignment in basic blocks. Angr converts a loop-free sequence of x86-64 instructions (i.e., a basic block) into an equivalent *logic formula* that represents all possible execution traces of the basic block. Utilizing a Satisfiability Modulo Theories (SMT) solver (e.g., Z3 [Res20]), the interpretation function checks whether the formula can be satisfied and if so, it determines the exact constant values assigned to the register state. Although angr uses Z3 to identify only one satisfying solution, instead of a complete set of all possible constant values, we ensure that the determined constant is indeed immutable via the process of elimination. Then, the

Algorithm 3: Interpretation function $\dot{Int}(e, C)$ that determines the register state for the given edge e by propagating constants and determining constant assignments to registers in the preceding basic block.

Input : Edge e and an annotated control-flow graph C
Output: Lattice element $post \in \dot{S}$

```
1  pre := NX;
2  foreach e' ∈ incoming(origin(e)) do
3      a := get_annotation(e', C);
4      pre := pre ⊔̇ a;
5  if pre = NX then
6      return NX;
7  S := angr.blank_state();
8  foreach r ∈ R do
9      if pre(r) ≠ X then
10         angr.constrain_register_value(S, r, pre(r));
11 F := angr.symb_exec_basic_block(origin(e), S);
12 post := λr ∈ R.r ↦ X;
13 if ¬z3.is_satisfiable(F) then
14     return NX;
15 foreach r ∈ R do
16     v := z3.get_model(F).get_value(r);
17     F' := F ∧ (r_end ≠ v);
18     if ¬z3.is_satisfiable(F') then
19         post(r) := v;
20 return post;
```

interpretation function propagates the lattice elements to the main abstract interpretation algorithm, which incorporates the results by annotating the edges, to finally identify the system call a distinct `syscall` instruction refers to.

To further clarify the details, Alg. 3 describes the interpretation function $\dot{Int}(e, C)$ that receives an edge e and an annotated CFG C as parameters. This function identifies the basic block, $origin(e)$, out of which the given edge e originates, and collects annotations of its origin's incoming edges. To propagate constants in the register state that precedes the basic block, $origin(e)$, the algorithm joins the annotations of its incoming edges in lines 1 to 4. This defines the initial (register) state of the basic block, which represents the precondition for the symbolic execution. Then, in lines 7 to 11, the interpretation function transforms the origin's initial (register) state into its symbolic representation S and adds constraints for each register $r \in R$ which holds a constant value $c \in \{0, \ldots, 2^{64} - 1\}$ at the

basic block's entry point (that expresses the constraint $r_{begin} = c$ at the basic block's entry point). At this point (line 11), we instruct angr to symbolically execute the basic block $origin(e)$, with the initial state S. Angr transforms the basic block into a logic formula F (that expresses the constraints given by the basic block). Once angr finalizes the formula, we use angr's SMT solver to determine whether or not F can be satisfied. If F cannot be satisfied, the control-flow will never reach the end of the basic block (given the constrained register state at the basic block's entry point). If F can be satisfied, we use angr's SMT solver to determine which registers hold a constant value after executing the basic block (lines 12 to 19). For each register, first, we add a clause to F that prohibits the assignment of the identified value v to the register at the end of the basic block. Then, we consult the SMT solver one more time to determine if the extended formula remains satisfiable. Only if the formula cannot be satisfied (due to the new constraint), we can assure that the basic block will always assign the same value to the register in question. At last, the interpretation function returns a lattice element, representing the register state after executing the basic block (i.e., the annotation for the edge e), to continue the abstract interpretation (Alg. 1).

7.3.3.3 Applying Abstract Interpretation

Given the right tools at hand (CFG C, Lattice $\dot{L}$, and $\dot{Int}(e, C)$), we are able to apply our abstract interpretation based constant propagation to identify the system call number that is passed to an identified `syscall` instruction in a binary. Once, the abstract interpretation has made a pass through the CFG, every edge in the CFG will be annotated with a lattice element representing the register state after each basic block. This includes the artificially included edge e' that immediately precedes the identified `syscall` instruction (Sec. 7.3.2). That is, to determine the system call number for a particular `syscall` instruction, we have to examine the annotation of e' and extract the final value from the `rax` register. In case `rax` maps to a constant value (`rax` $\in \{0, ..., 2^{64} - 1\}$), we can be sure that the `syscall` instruction will always receive the same system call number. Otherwise, if `rax` maps to X, we will not be able to deduce the system call number. In this case, we inform the analyst by stating the incompleteness of the analysis; she will have to complete the result before generating the seccomp filter.

7.3.4 Refining *seccomp* Policies

Exploiting a vulnerability in a system call in the Linux kernel can escalate a user-space application's privileges or escape from a container. To mitigate this threat, we propose using seccomp to enforce the *principle of least privilege* and prohibit the invocation of any system call that is not necessary for the genuine execution of the application. Consequently, we leverage Jesse to generate seccomp filters for programs and Docker containers. Disregarding whether an application is containerized or not, we first determine all dependencies, composing a set of binaries required by the individual ELF file or the container image; while we can simply inspect the ELF header to spot all libraries required by non-containerized applications, we have to first dissect container images to identify all binaries that reside in container images (Sec. 7.4.1). Then, we apply the introduced abstract interpretation to the discovered binaries to determine the set of system call numbers essential for their execution (Sec. 7.3). Finally, we generate a seccomp policy which whitelists the identified system calls; and blacklists the remaining system calls.

Unfortunately, without any additional steps, this naive approach tends to create coarse seccomp policies, as it does not consider that a system call (e.g., inside a library) may not be called; the corresponding system call invocation can be unreachable or simply not required by the binary. This presents an issue in particular for the standard C library, `libc`. In fact, our evaluation has shown that employing this naive approach would create seccomp profiles, which in case of the GNU `libc` implementation v2.28 would grant 273 system calls that stem only from the `libc` (Sec. 7.5.3.3). Consequently, to identify the unreachable system call invocations, we apply library debloating on the `libc`. This improvement allows us to compile and apply an effective seccomp policy that forbids the program to execute unneeded and potentially vulnerable system calls.

One of the inherent properties of shared libraries is their general-purpose character; they are built to provide functionality *"for all intents and purposes"*. At the same time, due to this characteristic, libraries are often considered as *bloated* [QPY18, AJWK+19, WKKWK+20], especially in the face of individual programs that often resort to only a tiny fraction of the entire library. This is in particular true for the standard C library (i.e., `libc`). Recent research has shown that 2016 Ubuntu applications make use of merely 5% of the standard C library (i.e., `libc`) on average [QPY18]. Similar to library debloating frameworks [QPY18, AJWK+19, WKKWK+20], we consider such fully-blown libraries as security threats, as they can govern the access to unneeded (and potentially vulnerable) system calls that the target program binaries never intended to use (in a benign setting).

The reason for this verdict is that `libc` represents the main gatekeeper between the system's user space applications and the kernel; it implements wrappers for the most system calls [Ker13a]. As part of the Linux Standard Base (LSB) [Fou15a], `libc` establishes a level of portability between distributions and architectures, and, more importantly, can be used by proprietary, closed-source applications. Consequently, if we determine all `libc` code regions that are unreachable by the extracted, benign program binaries and exclude these from Jesse's system call number identification, we will manage to create a framework

that generates high precision seccomp policies. We choose `libc` as the only library, in which we eliminate the unreachable code, as the state-of-the-art debloating techniques have requirements that are not necessarily met by closed-source binaries (thus eliminating a generic solution); they require recompilation [QPY18], meta information in unstripped binaries [AJWK+19], or must be position-independent [WKKWK+20, DWKJ+20].

Recent advances in code debloating on source code level [QPY18] as well as unstripped [AJWK+19] and stripped binary level [WKKWK+20] have demonstrated their effectiveness and precision in locating unreachable code in libraries. We have independently implemented an approach similar to Nibbler [AJWK+19], which we use to complement Jesse by identifying unreachable code in libraries with help of relocation information (as we do not claim any novelty, we refer the reader to Nibbler [AJWK+19]). In a general scenario, we assume a given (blackbox) framework (e.g., Nibbler) that allows us to locate the code regions associated with each exported library function. We highlight that this step must be done *only once*, either by the security analyst or by the library maintainer. We use the isolated code regions as input for our abstract interpretation to identify and create a (per function) map holding the involved system call numbers. This way, we can query the map to identify all system calls that are associated with the utilized exported library functions that are required by a given program binary (i.e., the dissected container service). Note that we do not regard this code reachability information delivery as an integral component of Jesse. Instead, we request this information from an external party only for one, central library, namely `libc`. Disregarding our decision for this work, the analyst can gather code reachability information for all involved libraries in order to generate even more concise seccomp policies. We leave this part for future work.

Once we identify the involved binaries (including programs, their libraries, and the libraries' unreachable code regions), we apply Jesse to (i) unfold their control-flows, leading to a set of `syscall` instructions and (ii) uncover the associated system calls (Sec. 7.3). Considering that most C programs employ the functionality of the standard C library, the net effect of our system call number analysis (without first eliminating the unreachable code in `libc`) would result in a highly coarse-grained and heavily overapproximated collection of system calls; `libc` implements an essential part of the OS interface including a high number of system call wrappers to simplify the communication with the OS kernel. As such, by applying the unreachable code information (per exported library function) Jesse considers only viable paths in `libc` in its system call number analysis.

7.4 Use Case: Enhancing Docker Container Security

To demonstrate the effectiveness of our system, in this use case, we leverage Jesse to generate seccomp filters for Docker containers. For each program in the container, we apply the introduced abstract interpretation to identify the system calls that are essential for its execution (Sec. 7.3). This set of system calls allows us to compile and apply an effective seccomp policy that forbids the container from executing unneeded and potentially vulnerable system calls. The quality and strength of the generated seccomp policies highly depend on (i) the precision and coverage of our static analysis (Sec. 7.3) and (ii) the accuracy of the optimization to disregard unused code sequences (Sec. 7.3.4). To determine both, we apply Jesse to five of the most popular Docker containers from Docker Hub [Doc20b].

7.4.1 Dissecting Docker Containers

Docker uses a *Dockerfile* (a structured document with instructions) to build containers. Among others, this file specifies the the main program to be executed inside the isolated environment via the `ENTRYPOINT` (or `CMD`) command. Although it is advised to provide only one service per container, it is not strictly prohibited to share one container among multiple services. Even if a container comprised only one service, its setup could require additional programs, e.g., to prepare the environment (i.e., switch to another user or initialize a database). That is, the specified executable can engage multiple programs in the Docker image, with each requiring its own set of libraries. Unfortunately, Docker images are known to incorporate a significant amount of unneeded programs and libraries [RNMJ17, RDDC+17]. Consequently, before Jesse determines the set of system calls required for the container, it identifies all programs that are necessary to prepare and provide the intended service(s). Specifically, Jesse implements existing container debloating techniques (without claiming novelty results) [RNMJ17, RDDC+17] to dissect container images and precisely narrow down the set of binaries to receive accurate results in the following steps.

7.4.2 Abstract Interpretation

We apply Jesse to five of the most popular Docker containers [Doc20b].[5] We used Jesse to dissect the containers to identify only necessary programs and libraries (Sec. 7.4.1). Table 7.1 summarizes and assigns the extracted binaries—holding at least one `syscall` instruction—to the associated container. By applying Jesse to the identified binaries, we were able to link the exact system call number to 719 of 725 `syscall` instructions in the containers' latest version. Jesse was not able to determine the system call numbers of the remaining 6 `syscall` instructions: 3 in `libc`, 2 in `libpthread`, and 1 in `mysqld`. Hence, in 99.17% of all cases, our analysis determined the correct system call (verified by manual

[5] For purposes of comparison, we selected the same set of containers as Wan et al. [WLX+17].

inspection). Similar to the results in Sec. 7.5.1, we do not consider the remaining 0.83% as false negatives. The reason for this is that, even though Jesse was not able to derive the system call number in question, it pointed us to the exact location of the troubling `syscall` instructions and requested our assistance to complete the analysis manually. We discuss and assess the reasons for the incomplete mapping in Sec. 7.5.2.

7.4.3 Avoiding Unreachable Code

We applied the discussed optimization that allows Jesse to focus only on viable paths in the `libc` in order to avoid considering system call invocations that remain unreachable to the program in question (Sec. 7.3.4). One of the benefits of this optimization is that it needs to be performed once per library version; for every binary that leverages services of the `libc`, Jesse queries the (per function) map of system call numbers required to tailor the seccomp policy. Specifically, we have applied Jesse to the previously selected set of containers, considering the additional (per function) code region information. (Note, while we have used a custom implementation to identify the binary code regions of exported functions in the `libc`, we can gather this information from external sources (Sec. 7.3.4).)

The optimization drops the `syscall` instructions in container binaries to a subset reached by the analyzed program. This way, we establish the basis for accurate and effective seccomp policies. To accommodate closed-source binaries, we apply the optimization only to `libc`; system calls of other dissected container binaries are accumulated through the general system call number analysis without the additional optimization (Sec. 7.3). In other words, we trade accuracy for compatibility and still achieve more accurate results than Docker's default seccomp policy. Note, contrary to library code debloating strategies [QPY18, AJWK+19], Jesse does not need to modify the libraries' layout in the target process' address space or code pages to remove the unneeded code at load-time, and hence does not exhibit any increased load-time performance or memory overhead.

Table 7.2 shows for each Docker container image the percentage of the restricted system calls through generated seccomp policies. On average, the generated policies restricted 54.7% of all system calls and significantly improved the granularity of Docker's default seccomp policy, which prohibits, depending on the container configuration, 10.6% to 20.4% of the system calls (Sec. 7.1.1). To rule out false negatives (i.e., falsely restricted system calls), we applied these policies to the Docker containers and ran benchmarks, testing the containers' functionality. We selected the same benchmarks used by Wan et al. [WLX+17], who, contrary to Jesse, use their work to *dynamically* mine seccomp policies. Instead of collecting performance results (as we did not modify the applications themselves), we focus on stressing the containers to achieve high coverage of the tested applications.

We employed `httperf` [MJ98] to test the genuine execution of the `nginx` and `httpd` web servers. In this context, we created 10 times 100 connections, each with an increasing connection rate from 5 to 50 requests per second (req/sec), with steps of 5 req/sec and 2 seconds sleep time whenever the connection rate is increased. Further, we tested the *mysql* containers by applying the benchmark `sysbench` [ako20]. We used `OLTP` test with 8

Table 7.1: JESSE's system call number identification analysis applied to ELF binaries (with at least one `syscall` instruction) that were extracted from (two different versions of) five popular Docker containers. The total number and percentage of the correctly assigned system calls refers to the analyzed binary's version residing in the respective container version.

	Binary	Httpd		MySQL		Nginx		Postgress		Redis		# of syscall	% of Assigned
		2.4.23	2.4.41	5.7.13	8.0.19	1.11.1	1.17.9	9.5.4	12.2	3.2.3	5.0.2	Instructions	System Calls
Libraries	libc	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	417 / 404	98.08% / 99.26%
	libpthread	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	158 / 169	98.73% / 98.82%
	ld	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	36 / 36	97.22% / 100.00%
	librt	✓	✓	✓	✓			✓	✓		✓	29 / 29	100.00% / 100.00%
	libattr				✓							- / 12	- / 100.00%
	libnuma			✓	✓							9 / 9	100.00% / 100.00%
	libsystemd								✓			- / 9	- / 100.00%
	libstdc++			✓	✓				✓			3 / 6	100.00% / 100.00%
	libaio			✓	✓							5 / 6	100.00% / 100.00%
	libcrypt	✓		✓		✓						1 / -	100.00% / -
	libkeyutils							✓	✓			3 / 3	100.00% / 100.00%
	libcap-ng								✓			- / 2	- / 100.00%
	libcrypto				✓		✓		✓			- / 2	- / 100.00%
	libuuid	✓	✓									2 / 2	100.00% / 100.00%
	libgcrypt								✓			- / 1	- / 100.00%
	libk5crypto								✓			- / 1	- / 100.00%
	libselinux			✓	✓			✓	✓			1 / 1	100.00% / 100.00%
Programs	mysqld			✓	✓							8 / 17	100.00% / 94.12%
	redis-server									✓	✓	2 / 8	100.00% / 100.00%
	nginx					✓	✓					6 / 7	83.33% / 100.00%
	httpd	✓	✓									1 / 1	100.00% / 100.00%

parallel threads and the maximum number of requests capped to 800. For the `postgres` containers we applied the `tpc-b-like` and the `simple-update` test of the `pgbench` [Gro20] test suite for 60 seconds each. Finally, we applied *redis-benchmark* [Lab20] to the `redis` containers. In all cases, the containers ran through and were not interrupted by a falsely prohibited system call.

7.4.4 Withstanding Real-World Exploits

We assessed the added security of the generated seccomp policy by using real-world exploits against (i) the MySQL server v5.7.14 running inside a Docker container and (ii) the Linux kernel v4.13. The combined exploits transformed a vulnerable system call into an effective *write* primitive that allowed the subverted container to directly modify the kernel memory for malicious purposes.

Table 7.2: Restricted system calls through seccomp policies tailored for five popular Docker container images. The seccomp policies were generated (statically) by JESSE and (dynamically) through *mining* by Wan et al. [WLX+17].

Container	Mining [WLX+17]	JESSE
`httpd v2.4.23`	78.9%	57.3%
`httpd v2.4.41`	–	55.3%
`mysql v5.7.13`	69.7%	49.5%
`mysql v8.0.19`	–	41.2%
`nginx v1.11.1`	77.8%	65.1%
`nginx v1.17.9`	–	61.1%
`postgres v9.5.4`	71.4%	50.7%
`postgres v12.2`	–	42.9%
`redis v3.2.3`	78.6%	65.4%
`redis v5.0.2`	–	58.5%

7.4.4.1 Case Study

We leveraged the CVE-2016-6662 (i.e., SQL injection vulnerability of the MySQL server) to gain arbitrary code execution capabilities inside the container. Generally, once the attacker gains control over the container, we assume she will attack the Linux kernel to attempt to escape the sandboxed environment, as this will grant her the capability to control other containers and even the kernel itself. To evaluate this scenario, we used the exploit for the CVE-2017-5123 that was introduced into the Linux kernel v4.13 to simulate an attacker that attempts to escape the container. In this context, first, we reproduced the exploit of Chris Salls, the discoverer of the CVE-2017-5123, to bypass KASLR [Sal17]. As our container was initially not bound by seccomp, and hence did not block unauthorized system calls, we abused the vulnerable `waitid()` system call to establish an *arbitrary write* primitive into the kernel memory; by abusing the fact that the vulnerable `waitid()` system call handler missed the necessary `access_ok()` checks (that prevent the user-space argument `siginfo_t *infop` from pointing to unauthorized memory), we were able to write-access arbitrary kernel addresses. Since the `unsafe_put_user()` (and other) kernel helper does not crash the kernel when accessing invalid memory, we were able to fingerprint the kernel's address space to identify its exact mapping [Sal17], despite KASLR.

In the next step, similar to the original exploit, the gained *write* primitive enable us to perform the *ret2dir* attack [KPK14]. To conduct the attack, we identified a page in the kernel's *physmap* (i.e., a contiguous memory region that directly maps a part of, or even all, physically available memory into the kernel space) that is aliased with a user-space page controlled by us. Then, we injected a *fake* data structure (e.g., `struct file`) with function pointers, which we used to initiate the execution of a ROP chain in kernel space. In this way, we disabled the system's SMEP and Supervisor Mode Access Protection (SMAP) protection and granted privileges to the executing container process. As soon as we received sufficient privileges, we became able, for instance, to request `CAP_SYS_MODULE`

capabilities, allowing us to load kernel modules and thus control the entire system.

Even though the attacker managed to write to arbitrary kernel memory, our seccomp policy was able to eliminate the vulnerability. Note that the Docker container did not require the vulnerable system call for its genuine execution. As such, after applying the generated seccomp policy to the container, seccomp successfully suppressed the unauthorized invocation of the vulnerable system call and immediately terminated the container.

7.4.4.2 Impact

In addition, to touch upon the impact of Jesse, we queried for CVEs that involve vulnerable system calls, which would have been allowed by Docker's default seccomp policy of the analyzed containers. We identified 30 CVEs for `nginx`, 26 for `httpd`, 24 for `postgres`, and 19 for `mysql`. We highlight that all of these CVEs could have been obstructed by the generated policy of Jesse. Note that we have identified the CVEs based on their description; therefore, they are by no means complete.

7.5 Evaluation

To evaluate our work, we have implemented Jesse and applied it to all binaries of the Debian *buster* base image. Since we have covered the security of Docker containers in the previous section (Sec. 7.4), we do not focus on analyzing container images. Instead, the following sections assess Jesse's analysis techniques in general. To realize Jesse, we have extended the LWD disassembler to identify the `syscall` instructions in binaries according to Sec. 7.3.2. Further, we have implemented the introduced abstract interpretation based constant propagation to determine the system call number for every previously identified `syscall` instruction, which we discuss in Sec. 7.3.3. To consider only viable execution paths in the `libc`, we have lend Jesse the ability to debloat the standard C library (Sec. 7.3.4); our library debloating implementation is independent of, yet, similar to Nibbler [AJWK+19].

Our evaluation of Jesse assesses the precision and coverage of the employed abstract interpretation based constant propagation (Sec. 7.3). To additionally amplify on the benefits of Jesse over existing static or dynamic analysis based frameworks, we provide test cases, in which we compare the effectiveness of Jesse with the results of the existing solutions. Specifically, we highlight that Jesse's static analysis capabilities, which determine the set of system call numbers of arbitrary non-obfuscated binaries, (i) exceed the capabilities of dynamic analysis based solutions [WLX+17], and (ii) complement previous static analysis frameworks [DWKJ+20, GPBP20].

7.5.1 Precision and Coverage

To evaluate the *precision and coverage*, we have applied Jesse to $1,064$ binaries of the Debian `buster` *base* image to identify system calls and link them to their respective system call numbers.[6] In addition, in order to establish a reference and to verify our assumptions, we have manually inspected all binaries holding at least one `syscall` instruction; our analysis has revealed that only 78 (out of $1,064$) ELF binaries hold at least one `syscall` instruction. Finally, we have compared our results by applying the static analysis based system call extraction framework, Confine [GPBP20], to the same set of binaries.

Table 7.3 shows that we have *manually* identified 984 distinct system call invocations, to which we have matched the associated system call numbers. By applying Jesse to the given set of binaries, we have identified 100% of the manually identified system call invocations and linked the exact system call number in 964 of 984 cases. Hence, in 98.0% of all cases, our analysis determined the correct system call (verified by manual inspection). We do not consider the remaining 2.0% as false negatives. This is because in the remaining 20 cases, Jesse pointed to the exact location of the troubling `syscall` instructions and requested our assistance to complete the analysis manually. This is a realistic strategy, as application developers or security analysts should create the seccomp policies.

[6] Used Debian mirror state from the 3rd of October 2020.

In addition, we have *manually* investigated how many of the $1,064$ binaries comply with **H1** – **H4**. Even though all of the analyzed binaries adhere to the x86-64 calling conventions, we were not able to (fully) linearly disassemble 7 (0.66%) binaries. The reason is that these binaries intertwine *data* with *code* in their `.text` sections, and thus violate (**H1**). We have encountered 10 (1.02%) system call invocations which violate (**H2**). We have not observed any violations of (**H3**). Jesse has informed us about each of the above violations; once we have resolved the violation, we were able to resume Jesse to complete the analysis. We clarify the reasons for the incomplete mappings of system calls in Sec. 7.5.2.

While the results of Jesse matched the number of the previously identified system call invocations, surprisingly, Confine has reported additional 158 system call invocations and flagged 183 cases for manual inspection. We have investigated this issue and identified that Confine (i) can falsely interpret byte sequences in the `.data` section as `syscall` instructions, and (ii) considers calls to all functions, whose symbols contain the string "`syscall`", as system call invocations. In addition, even though Confine's system call number approximation is conservative, it can miss system call invocations in case of tail-call optimizations. For instance, Confine will miss system call invocations, if the compiler replaces a `call` instruction to a generic system call wrapper (e.g., to the `syscall()` function in `libc`) with a `jmp` instruction to the respective wrapper; Confine's implementation only considers `call`s, yet, disregards `jmp`s to such wrappers. Finally, we have discovered that Confine has misidentified (32) as well as missed available system call numbers (76), which effectively introduces false negatives. Overall, in 89.7% Confine links the correct system call number to one of the 984 available `syscall` instructions.

In contrast, our results have shown that Jesse has automatically identified the correct system call number in 98% of all cases. Further, Jesse did neither over- nor under-approximate any system call invocations in the given set of binaries. In addition, under the assumption that the analyst always correctly analyzes the flagged system call invocations, Jesse has not produced any false negatives. Overall, combined with Confine's capabilities in analyzing the standard C library, we believe that Jesse can significantly contribute to the automatic generation of seccomp policies for (containerized) applications.

Table 7.3: Identified system call numbers of $1,064$ binaries of the Debian `buster` *base* image. The summarized results have been gathered through manual inspection (acting as baseline), Confine [GPBP20], and Jesse, and present the precision and coverage of the respective framework.

	Manual	**Confine**	**Jesse**
Raw numbers:			
◦ Analyzed binaries	$1,064$	$1,064$	$1,064$
◦ Identified system call invocations	984	1142	984
System call number identification:			
◦ System calls flagged for manual inspection		183	20
◦ Falsely identified system call numbers	0	32	0
◦ Missed system call numbers	0	76	0
Overall result:			
◦ Correctly identified system call numbers	100%	89.7%	98.0%

7.5.2 Reasons for Incomplete Mappings

Table 7.3 shows that out of the $1,064$ analyzed binaries of the Debian `buster` *base* image, Jesse requested manual support in 20 cases. The reason for the incomplete mapping of the system call numbers is threefold and in line with our hypotheses **H1** – **H4** (Sec. 7.3.1).

First, there exist functions, e.g., `syscall()` in `libc`, which allow selecting arbitrary system calls by specifying the system call number in one of the function parameters—even though this practice is not portable, error-prone, and mostly applied for testing [Cor10]. Since the exact system call number depends on the calling function, analyzing `syscall()` alone cannot determine the provided value (violation of **H2**). We can address this issue by generalizing our definition of the abstract interpretation; instead of identifying the value in the `rax` register before the `syscall` instruction, we cause the abstract interpretation to identify the value in the register holding the function's parameter (with the system call number) immediately before calling the `syscall()` function. Note that the exact register depends on the called function.

The second class refers to the incompleteness of the `Capstone` disassembly framework. For instance, `Capstone` failed to disassemble rare floating-point instructions and to determine data in the `.text` section (violation of **H1**), and hence impeded further analysis. In fact, Jesse encountered 7 cases, in which `Capstone` failed to disassemble parts of the binary. We have extended Jesse to mark such binaries for manual inspection, in which the analyst has the possibility, e.g., to assist `Capstone` by explicitly stating whether the troubling section belongs to the `.text` or `.data` section. One can solve this engineering issue by extending the capabilities of `Capstone`.

Finally, a small class of functions reads the system call number from memory. Albeit this strategy does not violate any of our hypotheses (**H1** – **H4**), it impedes static analysis; static analysis cannot precisely determine the values that are read from memory. Nevertheless, contrary to dynamic analysis (which cannot guarantee that the dynamic tests cover all paths of the program [WLX+17]), in this and all upper cases Jesse narrows down the `syscall` instruction's exact location and lets analysts incorporate their knowledge to satisfy the policy. We further discuss these cases in Sec. 7.6.2.

7.5.3 Impact Evaluation

We position Jesse among three existing frameworks, which employ both static [GPBP20, DWKJ+20] and dynamic analysis [WLX+17] to extract the set of system calls a regular or containerized application is allowed to invoke. We demonstrate that the inherent properties of dynamic analysis approaches can lead to false negatives. Even though we believe that the selected static analysis frameworks are great candidates, we amplify upon their limitations and demonstrate in what way Jesse can complement their functionality.

7.5.3.1 Dynamic Analysis

Generally, dynamic approaches generate more rigorous policies. Yet, they highly depend on the achieved program coverage by the employed test suites, and hence merely underapproximate the set of system calls that is vital for the analyzed program. In other words, in case a benchmark misses an execution path (that leads to a system call) in the target application, a potential execution of the hereby introduced false negative (a falsely unauthorized system call in the seccomp policy) will eventually falsely crash the process. To back our claims, in the following, we present pitfalls of the incompleteness of dynamic analysis based related work [WLX+17].

By closely analyzing the Docker container versions and the respective seccomp policies that were dynamically generated by Wan et al., we discovered that their analysis overlooked a set of essential system calls. For instance, the dynamically generated seccomp policy for the `redis` v3.2.3 container missed the system calls `rename()` and `fsync()`; both are used for the `redis`' background saving functionality. Another example is that the generated policy for the `nginx` v1.11.1 container missed the system calls `rename()` and `chmod()`; the system calls that define the web server's capabilities with regard to UNIX-domain sockets. These findings are incomplete, yet, they question the credibility of the dynamically gathered results and, at the same time, underline the need for a more complete, static analysis.

7.5.3.2 Static Analysis

Confine [GPBP20] focuses on identifying legitimate system call numbers from containerized (i.e., Docker) applications to establish seccomp policies for Docker containers. To achieve this, Confine requires the source code of the standard C library, `libc`. Confine's

analysis provides an accurate mapping of system calls to the exported functions of the `libc`. This allows the authors to map the system calls of the functions in the `libc` to binaries, which use the respective library functions. Unfortunately, the analysis of the remaining system calls (residing outside the `libc`) lacks precision. In fact, we have shown in Sec. 7.5.1 that Confine can produce false negatives, which are intolerable in production systems. In addition, Confine uses a backpropagation-only based technique, which, by design, cannot deal with loops.

Listing 7.1: Simplified code snippet from `libc` highlighting the limitations of pure backpropagation based methods.

```
    mov rbx, <syscall number>
    mov rcx, 0x10

begin_loop:
    mov rax, rbx
    syscall
    dec rcx
    jnz begin_loop
```

Listing 7.1 shows a simplified real-world example, which can be found in similar form in the `libc`. In the corresponding C code, the system call number is assigned inside the loop, immediately before the `syscall` instruction. Yet, the compiler has pulled the assignment of the constant system call number to a temporary register, `rbx`, in front of the loop to optimize the performance. Such cases render pure backpropagation-based methods as incapable of identifying the system call number. In contrast, Jesse is not limited by loops and does not require external assistance to identify the system call number in such cases.

On the other hand, `sysfilter` [DWKJ+20] provides highly precise results, as it additionally identifies and eliminates system calls in libraries, which are linked against binaries, yet, not needed at run-time. Unfortunately, `sysfilter` only works with PIC binaries, which is a limiting factor of the analysis. For instance, 43.9% of all binaries in the Debian base installation comprise dynamically-linked, non-PIC binaries, and hence should not be disregarded. Consequently, we regard the techniques applied in Jesse (Sec. 7.3) complementary to both Confine and `sysfilter`.

7.5.3.3 Library Debloating

As discussed in Sec. 7.3.4, in order to identify unreachable code we apply library debloating techniques *only* to the standard C library, `libc`. This strategy allows Jesse to consider only viable paths during the system call number analysis. To quantify the impact of this preprocessing step, we have applied Jesse to the GNU `libc` v2.28 that is part of the Debian base installation. It turns out that the library holds 404 `syscall` instructions, which call

273 distinct system calls. In other words, without applying debloating techniques, the resulting seccomp profile would, by default, grant 273 distinct system calls in addition to the ones that are not part of the `libc`. As such, as opposed to the resulting seccomp policies for the analyzed container images in Sec. 7.4.3, without this additional pre-processing step the restrictions would be less effective. For instance, the generated seccomp policies for the containers Httpd, MySQL, and Nginx would restrict only 17.3%, 14.1%, and 17.3% of all system calls without debloating the `libc`, instead of 55.3%, 41.2%, and 61.1% when debloating the `libc` respectively.

7.6 Discussion

Only reliable analysis results can be of value in production systems. As such, in this section, we assess the soundness of Jesse according to our key hypotheses (**H1** – **H4**), the compliance of which presents a *sufficient condition* to establish sound results in practice. We conclude this section by discussing the limitations of our work.

7.6.1 Soundness

In essence, Jesse comprises a *CFG construction* (Sec. 7.3.2) and an abstract interpretation based *constant propagation* technique (Sec. 7.3.3). Since our constant propagation strategy relies on the constructed CFG, we have to evaluate both in combination. Our implementation of the constant propagation leans on the work of Kildall, for which the author provides a proof of soundness [Kil73]. Kildall's proof bases its assumptions on that the CFG is a *correct* and *complete* representation of the analyzed program. Yet, since the CFG constructed by Jesse is *correct*, but not necessarily *complete*, we have to pose further requirements on the analyzed binaries to ensure the soundness of Jesse's results (Sec. 7.3.1).

First, we assume that the binary satisfies **H1**. **H1** assumes that benign and untampered, `gcc` and `clang` generated binaries can be linearly disassembled [ACvdV+16]. This lends Jesse the ability to (i) identify all `syscall` instructions in the binary, and (ii) ensure that the disassembled instructions in the basic blocks of the CFG correspond to valid basic blocks in the analyzed program. Furthermore, due to our conservative CFG construction mechanism, we can guarantee that all edges in the CFG represent valid control-flows in the analyzed binary. Also, **H1** safely assumes that the analyzed binaries adhere to the calling conventions of the x86-64 ABI. Specifically, Jesse assumes that a function's *caller* can only observe changes in *caller-saved* general-purpose registers; *callee-saved* registers are restored on every function return; violating this assumption would lead to false results if, e.g., a function call—prior to a `syscall` instruction—would not restore a callee-saved register's value, which is involved in the computation of the system call number.

In addition, we assume that every system call invocation in the target binary adheres to **H2** or **H3**. Generally, there exist only two analysis outcomes: either Jesse manages to identify a system call number for a specific `syscall` instruction or requests the analyst's assistance. Assuming Jesse identifies a system call number. If **H2** holds, the identified system call number is the only system call number that can be passed to the `syscall` instruction under analysis. If **H2** does not hold for this particular `syscall` instruction, yet, **H3** does, we can be sure that indirect jumps (inside of the function) do not affect the computation of the system call number. In other words, the incomplete CFG does convey the necessary information for Jesse to derive that a `syscall` should in fact be able to call different system call handlers. In such cases, the analysis either directly assigns X to `rax` as soon as it identifes a second possible value, or it propagates X from one of the function arguments to `rax` at the `syscall` instruction. In both cases, Jesse informs the analyst, regardless of whether or not the function itself has been called indirectly. (Note that if the

provided CFG is not correct, **H1** and **H2** will not suffice for sound analysis results. As such, we have ruled out heuristic-based CFG construction tools for Jesse—heuristics can either miss code or introduce illegal edges (Sec. 7.3.2).)

Finally, **H4** bases upon the observation that most binaries do not call system calls directly, but rather use wrappers in the standard C library. While this observation does not directly contribute to the soundness of Jesse, it does entail one corner case, which deals with *generic* system call wrappers, such as the `syscall()` function in `libc`. Jesse can determine the system call number provided to such wrappers, only if they are called directly. In such cases, Jesse considers calls to *generic* system call wrappers as hypothetical invocations of `syscall` instructions, which receive the system call number in another register (e.g., `rdi`), instead of `rax`. Yet, if a binary calls *generic* system call wrappers indirectly, Jesse will not be able to identify these calls and hence to derive the system call number. We have not encountered a single case of this type in the evaluation.

7.6.2 Limitations

Identifying semantic properties of programs are generally *undecidable*. Consequently, the employed abstract interpretation of Jesse only approximates the results and, in some cases, encounters its limits. These limits stem from both the dynamic nature of program behavior and the implementation deficits of our prototype, which we outline in the following.

Conditional branches: Jesse does not differentiate between the various branch targets to avoid having to determine the conditions that are only computed at run-time. Hence, if `rax` depended on a condition, our prototype would not be able to determine its value. Interestingly, we have not encountered any conditional assignments of the `rax` register that preceded a `syscall` instruction.

Generic system call wrappers: The scope of our constant propagation is limited to function-level. While this is sufficient for most system call invocations, there exist functions that act as generic wrappers (e.g., the `syscall()` function in `libc`) for arbitrary system calls; they allow to specify the system call number in one of the function's parameters. Since the exact system call number depends on the calling function, analyzing the `syscall()` function alone is insufficient to determine the provided system call number. We counter this issue by, first, letting Jesse inform us about the non-identified system call number, and second, instead of identifying the value in the `rax` register just before the `syscall` instruction, we identify (in the case of the `syscall()` function) the value of the `rdi` register just before calling the `syscall()` function. As this solution targets the `syscall()` function, a generic solution would need to consider the register state transferred between function calls. Yet, by analyzing binaries of Debian `base`, we have observed that `syscall()` is the only function that receives a system call number as argument (Sec. 7.5.1).

System call numbers in memory: Jesse's system call number analysis (Sec. 7.3) hinges on the premise that system call numbers are passed as constants via the `rax` register, and that they are not read from memory. Yet, some functions do not adhere to this assumption. For instance, `sighandler_setxid()` in `libpthread` reads the system call number from memory to determine whether to change the *user* or the *group id*. In such cases, Jesse informs and requests the analyst's input to make an informed decision.

Dynamically loaded code: C/C++ programs can constrain Jesse's analysis by either loading modules at run-time (via `dlopen()`), or dynamically generating code (via JIT compilation). To address the former case, we rely on the analyst to instruct Jesse to analyze all binaries loaded at run-time. Yet, static analysis cannot make any assumptions about the dynamically generated code.

Attack vector reduction: Although seccomp effectively reduces the system's attack vectors, it cannot completely diminish future attacks. Jesse (as the last line of defense) is most effective combined with other security hardening measures. For instance, when combined with accurate library code debloating mechanisms [QPY18, AJWK+19, WKKWK+20], Jesse would establish a restricted environment, deprived of unnecessary and potentially threatening gadgets and system calls.

Considering k-set domains: Jesse's abstract interpretation based constant propagation considers an abstract domain that restricts the number of constant values in registers to *one*. In other words, our analysis cannot determine a *set of different* constant values, a register can hold at a specific point. Instead, we assign $\top$ to the register if it can hold more than one value (Sec. 7.3.3.1). This implementation is in line with **`H3`**, which assumes that each `syscall` instruction is dedicated to invoke *only one* system call handler. We can address this restriction by basing the abstract domain on k-sets to allow registers to hold up to k different values. In this way, our analysis would be able to determine more than one system call number per `syscall`. Interestingly, we have not encountered a single case, in which this extension alone would improve the results in Sec. 7.5.1.

Virtual Dynamic Shared Object: On Linux, the `vDSO` is an architecture-dependent, shared library mapped into the address space of every user space process [Bov14b]. Generally, the `vDSO` increases the performance of selected system calls by emulating their functionality in user space; virtual system calls do not have to switch into the kernel space to perform their task. While the `vDSO` is practical, its architecture dependencies can introduce difficulties to dynamically generated seccomp policies [Cor19]. For instance, the time-keeping system calls highly depend on the program's choice of the hardware's clock source and the system's configuration [Cor19]. In case the `vDSO` (or rather its virtual system call `gettimeofday()`) should not be in the position to meet these demands, it will fall back to calling the system call in kernel space. Note that before Linux kernel v5.3, the fall back called the 32-bit `clock_gettime()` system call [Cor19]. This introduces an issue for both static and dynamic approaches for generating seccomp policies. Static approaches cannot analyze the `vDSO`, and hence miss system calls that are invoked in the respective fall back paths of virtual system calls. That is, even though the static analysis is able to extract the

called virtual system call (e.g., `gettimeofday()`), it will not identify the system call number that will be potentially called on another system (e.g., `clock_gettime()`). Similarly, dynamic approaches can miss the system call numbers used in the fall back path as the system could be differently configured on a different machine. To address this issue, we have completely analyzed the `vDSO` and, by default, authorize the execution of all system calls that can be called by the `vDSO`.

7.7 Related Work

In this chapter, we have presented Jesse, a framework for creating seccomp policies to establish a safe, sandboxed environment for Docker containers. In this context, we have combined different techniques, which are related to sandboxing, debloating, and constant propagation. In the following, we relate our work to selected research contributions.

There exists a significant body of research dedicated to analyzing and sandboxing applications and their libraries. For instance, Janus [GWT+96] is one of the first sandboxes for untrusted applications. It uses the Solaris' dynamic tracing capabilities to interpose and confine system calls. Automatically generating system call whitelists has been also subject to Host-based Intrusion Detection Systems (HIDSs). For example, Wagner et al. [WD01] and Feng et al. [FGH+04] use system call traces to detect malicious behavior. They apply static analysis techniques to construct automatons representing the benign system call traces and use them to detect malicious behavior at run-time. However, both approaches rely on hand-crafted (i.e., expensive and error-prone) models to identify the set of system calls of library functions. Thus, Jesse can complement their approaches.

Enhancing security of Android applications: Further sandboxing techniques have been applied to Android. Boxify [BBH+15] establishes a framework that uses Android's process isolation to execute apps in the context of a trusted process with restricted permissions. It limits the capabilities of untrusted apps by intercepting system calls and calls to the Android API. DroidSafe [GKP+15] models the Android API and uses static analysis to reveal information leaks in Android applications. Similarily, FlowDroid [ARF+14] is a static taint analysis that offers higher precision. While these approaches implement the necessary means for isolating and restraining individual applications, they cannot determine relevant permissions and system calls that should be whitelisted. As such, Jamrozik et al. [JvSRZ16] introduce a dynamic analysis based framework, coined sandbox mining, to determine the resources required by applications at run-time. They present boxmate, a framework that executes tests against target applications on Android to extract the set of resources (including system calls) required during the tests. This allows boxmate to limit the applications to the gathered resources. Instead of relying on conventional benchmarks, DroidBot [LYGC17] generates custom tests utilized by the framework of Le et al. [LBL+18], to also consider parameters of calls to the Android API.

Enhancing security of Linux containers: Wan et al. [WLX+17] mine sandboxes for Linux containers. They apply dynamic analysis to determine authorized system calls of Linux binaries. Similar to Jesse, they use the gathered information to generate and apply `seccomp` to filter unauthorized system calls. Unfortunately, due to the inherent properties of dynamic analysis, their framework can miss system calls. Similar to Wan et al.'s approach, Ghavamnia et al. [GPBP20] also construct seccomp profiles based on the results of a static source code analysis called Confine. A novel mechanism is the temporal system call specialization by Ghavamnia et al. [GPMP20] that divides the execution of a program into phases and applies different seccomp filters in each of these.

Container and library debloating: Another research direction applies debloating technique to containers. Rastogi et al. [RDDC+17, RNMJ17] develop Cimplifier, a framework that partitions, or distributes applications inside a container into multiple, minimal, isolated, and interconnected containers. Their research, applies either dynamic [RDDC+17] or static analysis [RNMJ17] techniques to discover essential and to remove unused programs from the container's image. In this way, Rastogi et al. manage to eliminate unneeded bloat from container images. Other debloating techniques aim to eliminate unneeded code in program libraries. Quach et al. [QPY18] statically analyze the binaries to discover unneeded instructions in libraries. They use the gathered information to unmap irrelevant code in the process' address space. Additionally, Quach et al. overwrite irrelevant code in the process' address space on sub-page granularity by relying on the copy-on-write mechanism. Yet, the authors base their analysis on the intermediate representation of LLVM and hence they have to recompile the analyzed binaries with `clang` (which is not able to compile `libc`). Agadakos et al. [AJWK+19] develop a binary-level library debloating tool, Nibbler, which analyses binaries to identify and erase unused functions in libraries, without having to recompile the target binaries. Yet, the presented tools rely on additional meta information in unstripped binaries, gathered from the ELF symbol table. Egalito [WKKWK+20] is a framework for recompiling binaries to enhance their security (e.g., by reordering instructions, introducing retpolines, JIT-shuffling, etc.). In addition, similar to Nibbler, Egalito applies techniques to debloat programs and libraries. Contrary to Jesse, Egalito requires binaries to support PIC to gather sufficient metadata. Although Egalito significantly reduces the code size by recompiling the binaries, it cannot remove code that is not executed due to disabled features in the configuration; such code is only dead in the current configuration but not in general. As response, Koo et al. [KGP19] develop a configuration-driven software debloating technique that overcomes this limitation. Koo et al. analyze which libraries are only used to provide one specific feature and remove these libraries when the feature is disabled.

Constant propagation: Finally, constant propagation is a well-studied research direction. Kinder et al. [KV08, KZV09, KV10], Fleury et al. [FLPV15]), and Bardin et al. [BHV11] employ constant propagation to recover CFGs from binaries. While Bardin et al. rely on symbolic execution to underapproximate the CFG, Kinder et al. and Fleury et al. overapproximate the CFG by using abstract interpretation identifying possible targets of indirect branches.

7.8 Summary

Contrary to the security hardening techniques of previous chapters, in this chapter, we have explored our final objective, which deals with the question how to assist OS-level virtualization (Q3). Instead of relying on hardware virtualization extensions, we have utilized the Secure Computing (seccomp) mode that is available to the Linux kernel. The presented techniques do not require a dedicated VMM to enforce the security guarantees, yet, they can complement orthogonal mechanisms that have the ability to further increase the isolation of containers, e.g., through virtualization (Chap. 6). That said, we have introduced Jesse, a framework that leverages static analysis to tailor seccomp policies for ELF binaries and Docker containers on Linux. Specifically, we have extracted sandboxed ELF binaries from container images and utilized an abstract interpretation based constant propagation to identify the system calls in the extracted binaries. More precisely, we have analyzed arguments of `syscall` instructions to determine a set of authorized system call numbers. We have further combined our analysis with state-of-the-art library debloating techniques to narrow down the necessary code regions associated with each exported function in the standard C library, `libc`. We use the isolated code regions in the `libc` to create a (per function) map with the involved system call numbers. In this way, Jesse can query the map to identify the system calls of an exported function that is required by the analyzed binary. Once we have extracted the system call numbers, we have compiled seccomp policies for the respective Docker containers to thwart accesses to unnecessary, and potentially vulnerable system calls. Finally, we have demonstrated the effectiveness of our work by applying Jesse to all binaries in the Debian *base* image and popular Docker containers, and by reinforcing seccomp against real-world container escalation exploits. Overall, the results in this chapter confirm that Jesse can be the last line of defense against real-world exploits.

Chapter 8

Conclusion

Prediction is very difficult, especially about the future.
— NIELS BOHR

The technological advances and the adoption of virtualization has shown great potential with regard to OS security. During the past two decades, virtualization has undergone a paradigm shift, in which it has extended its horizon from virtualizing servers towards OS security. Chen et al. [CN01] were among the first to identify the potential and challenges behind virtualization-assisted security. This, at the time novel direction has set an incentive to leverage virtualization to relocate security-related services out of the OS into an isolated environment. Similarly, the seminal paper of Garfinkel et al. [GR03] has formed the origin of virtual machine introspection. The concepts behind VMI have laid the foundation for various frameworks and continue to evolve up to this day. Both papers have strongly influenced and inspired many researchers to emphasize the value of virtualization technology on security. With time, security through virtualization has spilled over from academic circles. Today, it receives increasing acceptance from the public and industry sectors. For instance, Microsoft uses their VMM, Hyper-V, not only to virtualize servers but also to equip Microsoft Windows with various virtualization-assisted security policies. Overall, we believe that the capacity of virtualization in regard to security has not yet been explored to its full extent, which has become the main drive for our research.

8.1 Contribution

Inspired by previous research on virtualization, the technological progress, and their mutual drive, in this work, we have explored new ways of repurposing the hardware's virtualization extensions to pursue two main research directions: namely (i) novel virtualization-assisted techniques to improve the state-of-the-art VMI-based dynamic binary analysis frameworks and (ii) primitives to support the security of selected OS components (Sec. 1.2). To put the perspective on these two main research directions into a concrete frame, we have outlined an abstract target system architecture in Chap. 3. We have used the simplified target architecture to organize and position the concepts and intents of our contributions. Also, we have accompanied the simplified architecture with a general threat model, which defines offensive and defensive adversarial tactics to shed light on the presumed opponent.

Our line of research has begun by investigating on-demand VMM deployment strategies (Chap. 4). By adopting the deployment scheme of virtualization-assisted rootkits [Rut06b, RT07, Zov06], we have designed a thin, microkernel-based framework, WhiteRabbit. We have utilized WhiteRabbit to assist OSes to dynamically take control over the system's virtualization extensions, and to install arbitrary virtualization-assisted services according to (i) and (ii). We have demonstrated the potential of WhiteRabbit's deployment strategy by installing an exemplified VMI framework on a Linux system, on-the-fly. By following this line of research, we have gained a renewed perspective on the virtualization extensions on x86 and ARM; we were able to utilize their capabilities on-demand—i.e., without having to rely on a fully-fledged VMM, which typically has to be deployed before the OS itself. Yet, more importantly, through this framework, we have gained the insights necessary to approach the objectives that have helped guiding our research (Sec. 1.2).

The modern ARM architecture has become one of the most prevalent architectures for mobile, wearable, and IoT devices, and also has found its way into the server market [Ama20, The20b]. We have explored the ARM architecture to identify whether it is equipped with hardware capabilities necessary for stealthy VMI. Even though our investigations have revealed that ARM lacks the foundation required for stealthy monitoring, with Q1, we have set an objective to identify alternate ways to overcome this limitation. We have observed that we can compensate the hardware deficits, by introducing alternative primitives to empower stealthy monitoring on ARM. In Chap. 5, we have shown how these primitives can establish alternative ways of setting and single-stepping software breakpoints, without using the intended hardware extensions. In other words, we have repurposed the virtualization extensions on ARM to facilitate stealthy primitives for VMI. Specifically, these primitives leverage our implementation of the Xen `altp2m` subsystem, which utilizes the SLAT mechanism to define and switch among alternate views on the guest-physical memory. By configuring the alternate views in a sophisticated way, we manage to set and single-step over injected tap points in memory, without revealing them to the monitored VM. To demonstrate the effectiveness of our achievements, we have equipped the state-of-the-art dynamic binary analysis framework, DRAKVUF, with our primitives, and hence ultimately empowered ARM with stealthy VMI capabilities.

As part of our next objective (Q2), we have shifted the focus of our research away from VMI towards integrating the capabilities of hardware virtualization into Linux to enhance the security of its subsystems (Chap. 6). This strategy has allowed us to overcome the limited access permissions of the MMU in regard to memory isolation. We have introduced virtualization-assisted selective memory protection (xMP) primitives to counter data-oriented attacks in kernel and user space. Instead of outsourcing policies and knowledge to enforce isolation of selected memory regions to the VMM, we have empowered the memory management system of the OS with the ability to isolate sensitive data in disjoint xMP domains. The xMP primitives additionally equip pointers to sensitive data in kernel memory with context-bound authentication codes to prevent unauthorized pointer redirections. We have shown that Intel's in-guest EPTP switching capability perfectly suits our demands; this hardware extension allows VMs to take over some of the EPT management tasks, and hence to maintain different views on the guest's physical memory. At the same time, we have discussed that we can apply xMP, e.g., to the ARM architecture by implementing dedicated hypercalls. In both cases, by utilizing xMP, guest OSes do not have to relocate the defender's logic into the VMM. To assess our work, we have implemented and applied xMP primitives to protect all page table structures and process credentials in kernel space, and cryptographic material of selected user space applications. In addition, we have established the foundation for virtualization-assisted container security, by integrating xMP into Linux namespaces. In all cases, we have shown that the number of xMP domains scales and incurs only low performance overhead for real-world applications.

By having integrated xMP into the Linux namespaces, we have started to shift our focus towards our final objective Q3. In this context, the question we have asked ourselves is how to assist OS-level virtualization (i.e., containers) to fortify their isolation capabilities? Through xMP, we have shown that we can equip Linux namespaces—an essential building block of modern Linux containers—with enhanced memory isolation capabilities. The xMP namespaces leverage SLAT to isolate selected system resources. In this way, xMP provides a hardware-backed defense, which can be utilized by the Linux kernel to obstruct adversaries from corrupting or leaking the contents of sensitive kernel data structures, even if a malicious actor has gained primitives to arbitrarily *read* or *write* to kernel memory.

To approach Q3 from a different point of view, in Chap. 7 we have introduced Jesse, a static analysis based framework to tailor seccomp policies for Docker containers on Linux. Jesse filters not needed (and potentially vulnerable) system calls that would be otherwise freely available to the applications in the container. In this regard, Jesse (i) extracts relevant ELF binaries from Docker container images; (ii) applies its abstract interpretation based constant propagation to identify an over-approximated set of system calls, the respective binary is authorized to invoke; and (iii) compiles Docker-compatible seccomp policies. In addition, we have implemented common library debloating techniques to identify the system calls of each individual function of the standard C library, `libc`. This scheme has allowed us to create a map, which we can query to determine the system calls that are associated with each exported function in the `libc`. Finally, we have applied Jesse to popular Docker containers to protect them against real-world container escape exploits.

8.2 Future Research Direction

Virtualization technology has reached a state, in which it receives increasing acceptance from the public and industry sectors. Undoubtedly, this technology is indispensable in cloud environments, yet, it has also proven itself perfectly suitable in regard to security. In line with our work's research directions, in the following, we share our thoughts on our ongoing and future research directions on VMI-based dynamic binary analysis and virtualization-assisted OS security.

Selective driver protection: Linux device drivers (in form of loadable kernel modules) establish an interface between the OS kernel and physical devices. Being responsible for handling the devices' input, kernel modules face the risk of being abused by malicious actors, who aim to gain access to the most sensitive components of the OS kernel. To alleviate this issue, our ongoing research investigates *selective driver protection (xDP)* primitives for Linux. The general idea behind xDP is to prevent potentially vulnerable kernel modules from mounting code-reuse attacks against the Linux kernel. Specifically, we can use xDP to isolate selected device drivers and filter their communication with the kernel. This essentially divides the problem into two steps. First, we have to generate filters to grant access to only authorized kernel functions. We can implement this step as part of the module loading procedure (this is where the kernel resolves the addresses of the exported functions). Second, we have to isolate the drivers from the kernel through SLAT tables and mediate their control-flow transfers to the kernel by enforcing the (per-driver) filters.

Inspired by the system call filtering capabilities of seccomp on Linux (Sec. 2.1.2), our idea is to filter unauthorized control-flow transfers from a driver to the kernel, and hence thwart driver-originated code-reuse attacks against the kernel. The interaction between kernel modules and the kernel itself takes place through (i) exported kernel functions and (ii) dynamically initialized function pointers that point to the kernel's functionality. While we can address (i) by extracting the exported kernel functions that are required by the particular driver (by parsing the kernel module's relocation symbols), addressing (ii) requires further investigation, which we intend to pursue in our future work.

Once the filters are complete, much like with xMP (Chap. 6), we can utilize the EPTP switching and the `#VE` functionality on Intel to unmap the kernel's code segments (i.e., map the entire kernel memory as *non-executable*) from the driver's view and mediate the transitions. By creating two views on the guest's physical memory (a *driver view* and a *kernel view*, with both components being mutually *non-executable* in the respective opposite view), we can obstruct execution attempts of unauthorized functions. The driver will trap on every execution attempt of the *non-executable* kernel code into the `#VE` handler, in which we can switch the views to continue the execution. Since the `#VE` handler will be invoked on every transition of both views, it will allow us to apply the previously defined filter to grant transitions to only authorized functions. Similarly, when the kernel executes a driver's function, it will trap into the `#VE` handler, which will switch back to the *driver view*.

Hardening the Linux kernel slab allocator through in-place metadata isolation: Another research direction that we have started to investigate is the *in-place isolation of metadata* in slab-based dynamic memory allocators on Linux [Mom20]. At the heart of its memory management, the Linux kernel utilizes the slab allocator to facilitate fast and memory-efficient allocations of kernel objects (Sec. 6.4.2). The Unqueued Slab Allocator, `SLUB`, is the default slab allocator in modern Linux kernels. It maintains *slab caches* for frequently-used objects of the same size. More importantly, the `SLUB` allocator maintains allocated objects and metadata, e.g., of free objects in the same *slab cache*. In other words, both metadata, which is required for managing slab objects, and the allocated slab objects themselves share the same physical pages; this design choice lends adversaries sufficient ground for abusing the slab object's metadata to mount various attacks against the Linux kernel.

As a logical next step to our research on virtualization-assisted primitives (Chap. 6), this research direction leverages virtualization to provide a split perspective on slab objects to obstruct attackers from mounting attacks by abusing the heap's metadata. In particular, our ongoing research investigates a novel slab-hardening mechanism, which equips the slab allocator with the ability to separate the metadata from the user-controlled objects, without changing the virtual address layout of both elements. Specifically, our idea is to establish an *in-place isolation* scheme, which utilizes the system's SLAT mechanism to define separate views on the guest-physical memory. In this context, each view translates the object's address to a machine-physical address that holds either the object's data or its metadata. This way, the suggested slab-hardening mechanism associates every slab object with a view on its data (*data view*) and a separate view on its metadata (*meta view*). Thus, the system can configure the *data view* as default to prevent attackers from corrupting the metadata, and switch to the *meta view* only in controlled and safe locations.

Formal requirements for VMI-capable architectures: We have observed a cycle, in which hardware vendors announce and incrementally roll out novel hardware extensions, which are then explored by researchers with the intent to utilize the new hardware capabilities for their purposes. This process does not always fully meet the needs of the researchers, which are then forced (in the best case) to combine the hardware features with less-performing software techniques. Similarly, throughout our work, we have repurposed existing hardware features to meet our demands. Understandably, in only very rare cases, research—sometimes driven by the industry—can directly influence the hardware vendors to develop dedicated features. To counteract these conditions, researches can resort to an open source ISA, such as RISC-V, to implement the necessary features directly in hardware, and ideally introduce these features into the open standards that are maintained by the community. In this regard, we believe it would be of great value to the research community to specify and design the necessary hardware requirements for VMI (e.g., in form or a set of hardware-assisted capabilities) to define a dedicated VMI-capable architecture.

Virtualization-assisted OS architecture: We have started our work by exploring novel primitives to improve the capabilities of state-of-the-art VMI techniques. Over time, we have turned our attention towards assisting the security of OS architectures. In Chap. 6, we

have introduced our first attempt to integrate the capabilities of the system's virtualization extensions directly into the OS for defense purposes. Contrary to VMI, where external monitors utilize these capabilities to establish concealed monitoring environments, we envision a paradigm shift in the design of OS architectures. We believe that future OS architectures should alleviate the strict line between the OS and the VMM. Instead, we suggest that OSes should directly incorporate the system's virtualization extensions as inherent building blocks into their subsystems. We have observed that the prominent OS and VMM vendors have begun to utilize their virtualization to enforce security policies. Yet, we suggest to take this approach to the next level. We envision the next generation of OS architectures to involve and utilize virtualization extensions as an integral part of the OS. In other words, we consider the capabilities of virtualization extensions not limited to VMMs, but rather as a hardware resource, whose properties can greatly benefit the subsystems of the OS. The kernel can dedicate a kernel module to take control over these hardware features (Chap. 4). This scheme would allow different subsystems of the OS to utilize the added hardware properties to enforce custom policies, without relying on a fully-fledged VMM. Given that the technological state of modern hardware architectures supports nested virtualization, our idea does not conflict with multi-VM environments.

Virtualization-assisted container security: Further enhancing the security of containers is one of the next logical steps of our research. This research direction partially overlaps with the virtualization-assisted xMP primitives, which have allowed us to enhance the security of containers by extending the capabilities of the Linux namespaces (Chap. 6). Given the flexibility of modern containers, they became not only prevalent in hosting various services, but also a convenient environment for analyzing potentially malicious binaries [Sti20]. Hence, equipping the OS's namespaces with virtualization-assisted isolation features (as we have done with xMP) is a necessity in protecting against maliciously-motivated container escapes. At the same time, we can imagine scenarios, in which we can leverage the hardware-supported memory encryption features, combined with an alternate SLAT scheme, to enhance the security of containers. For instance, by exploring the announced Multi-Key Total Memory Encryption (MKTME) technology of Intel, future research should investigate the benefits and consider applying the novel features to use different keys to encrypt the memory of containers and the remaining parts of the OS.

By considering container-based analysis, we can think of methods that utilize the hardware's SLAT to establish stealthy analysis of binaries inside containers. Even though previous research on container-based malware analysis techniques has progressed, the applied methods lack a stealthy operation: both setting and single-stepping breakpoints can reveal the analysis framework. Thus, they face challenges which similarly occur in VMI-based analysis in regard to split-personality malware. To improve the state-of-the-art, similar to the presented techniques in Chap. 5, future research can consider alternative ways to repurpose the system's SLAT to implement stealthy monitoring primitives for containers.

8.3 Final Words

Overall, through our research, we have demonstrated that virtualization technology has a lot to offer. We have improved the stealthiness of state-of-the-art VMI-based dynamic binary analysis frameworks on ARM. In fact, to the best of our knowledge, we were the first to enable *stealthy* monitoring on ARM systems. In addition, we have provided an insight into our envisioned virtualization-assisted OS architecture that alleviates the strict separation between an OS and a VMM. The system's virtualization extensions can be utilized by the OS kernel directly to enhance the security of its subsystems, without having to deploy a fully-fledged VMM. In parallel, and entirely independent to our research, we have observed an industrial trend that follows a similar direction, which further supports our drive and motivation towards virtualization-assisted security. In conclusion, I strongly believe, and emphasize that the capacity of the virtualization technology in regard to security has not been explored to its full extent. As such, I look forward to investigating new avenues of virtualization that are yet to be discovered.

Glossaries

Symbols

BXJ Branch Exchange Jazelle.
EL exception level.
MTF Monitor Trap Flag.
PC Program Counter.
PL privilege level.
SMC Secure Monitor Call.
VMID Virtual Machine Identifier.
VTTBR Virtualization Translation Table Base Register.
#VE Virtualization Exceptions.
vDSO virtual Dynamic Shared Object.

A

ABI Application Binary Interface.
AMT Active Management Technology.
API Application Programming Interface.
ASLR Address Space Layout Randomization.

B

BOP Block-Oriented Programming.
BPF Berkeley Packet Filter.

C

CFB Control-Flow Bending.
CFG Control-Flow Graph.
CFI Control-Flow Integrity.
CPI Code Pointer Integrity.
CPU Central Processing Unit.

D

DEP Data-Execution Prevention.
DFI Data-Flow Integrity.
DKSM Direct Kernel Structure Manipulation.
DMA Direct Memory Access.
DOP Data-Oriented Programming.
dTLB data TLB.
DVM Dalvik Virtual Machine.

E

eBPF extended Berkeley Packet Filter.
ELF Executable and Linkable Format.
EPT Extended Page Table.
EPTP EPT pointer.

H

HCS Host Compute Service.
HIDS Host-based Intrusion Detection System.
HLAT hypervisor-managed linear address translation.
HLL High-level Language.
HMAC Keyed-Hash Message Authentication Code.

HVM hardware-assisted VM.

I

I/O Input/Output.
IDT Interrupt Descriptor Table.
IOMMU I/O Memory Management Unit.
IoT Internet of Things.
ISA Instruction Set Architecture.
iTLB instruction TLB.

J

JIT Just-In-Time.
JIT-ROP Just-In-Time ROP.
JVM Java Virtual Machine.

K

KASLR Kernel Space Address Layout Randomization.
KPTI Kernel Page Table Isolation.
KVM Kernel-based Virtual Machine.

L

LSB Linux Standard Base.
LSM Linux Security Modules.
LWD Lightweight Disassembler.
LXC Linux Containers.

M

MAC Mandatory Access Control.
MKTME Multi-Key Total Memory Encryption.
MMU Memory Management Unit.
MPK Memory Protection Keys.
MSR Model Specific Register.

N

NTP Network Time Protocol.

O

OCI Open Container Initiative.
OS operating system.

P

PAC Pointer Authentication Code.
PCB Process Control Block.
PE Portable Executable.
PIC position-independent code.

R

RCU Read-Copy-Update.
ROP Return-Oriented Programming.

S

SEV Secure Encrypted Virtualization.
SEV-ES SEV Encrypted State.
SEV-SNP SEV Secure Nested Paging.
SFI Software-Fault Isolation.
SLAT second level address translation.
SMAP Supervisor Mode Access Protection.
SMEP Supervisor Mode Execution Protection.
SMM System Management Mode.
SMMU System MMU.
SMT Satisfiability Modulo Theories.
SPP Sub-Page Write-Permission.
SSL Secure Socket Layer.
sTLB shared TLB.

T

TCB Trusted Computing Base.
TEE Trusted Execution Environment.
TLB Translation Lookaside Buffer.
TLS Transport Layer Security.

U

uniTLB unified TLB.

V

vCPU virtual CPU.
VM virtual machine.
VMBR virtual machine based rootkit.

VMCS Virtual Machine Control Structure.
VMI virtual machine introspection.
VMM virtual machine monitor.
VMX virtual machine extensions.

W

WSL Windows Subsystem for Linux.

Bibliography

[AA20] Samuel Axon and Ron Amadeo. This is Apple's roadmap for moving the first Macs away from Intel. https://arstechnica.com/gadgets/2020/06/this-is-apples-roadmap-for-moving-the-first-macs-away-from-intel/, June 2020.

[AB12] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: A Fast Short-Input PRF. In *International Conference on Cryptology in India*, 2012.

[ABEL05] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[ACvdV+16] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, 2016.

[Adv20] Advanced Micro Devices. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, January 2020.

[AJWK+19] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[ako20] akopytov. SysBench. https://github.com/akopytov/sysbench, June 2020.

[Ama20] Amazon Web Services Inc. AWS Graviton Processor. https://aws.amazon.com/ec2/graviton/?nc1=h_ls, June 2020.

[Ang20] Angr. Angr Documentation. https://docs.angr.io, June 2020.

[Apa19] Apache HTTP Server Project. ab - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/2.4/programs/ab.html, April 2019.

[App20] Apple. *Apple Platform Security*. 2020.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*, 2014.

[ARM14] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition (DDI 0406C.c)*. 2014.

[ARM19a] ARM mbed. mbed TLS Sample Programs. `https://github.com/ARMmbed/mbedtls/blob/master/programs`, April 2019.

[ARM19b] ARM mbed. mbed TLS v2.16.1 Source Code Documentation. `https://tls.mbed.org/api`, April 2019.

[Arm20] Arm Ltd. *Arm Architecture Reference Manual, Armv8 for Armv8-A Architecture Profile (DDI 0487F.b)*. 2020.

[Arm21] Arm. *Arm Confidential Compute Architecture Security Model, ARM-DEN-0096*. 2021.

[BBF+16] Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bülent Yener. AVLeak: Fingerprinting Antivirus Emulators Through Black-Box Testing. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[BBH+15] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-Fledged App Sandboxing for Stock Android. In *USENIX Security Symposium*, 2015.

[BBTV15] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro*, 2015.

[BC05a] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter Memory Management, pages 294–350. O'Reilly Media, 3^{rd} edition, 2005.

[BC05b] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter Interrupts and Exceptions, pages 131–188. O'Reilly Media, 3^{rd} edition, 2005.

[BCK+10] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2010.

[BCN+17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 2017.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review (OSR)*, 2003.

[BDOT16] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code. In *International Conference on Cyber Warfareand Security (ICCWS)*, 2016.

[Bed20] BedRock Systems Inc. BedRock Systems. `https://bedrocksystems.com/`, June 2020.

[BGRT05] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86–A Platform for Analyzing x86 Executables. In *ACM SIGPLAN Conference on Compiler Construction (CC)*, 2005.

[BHK+14] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[BHR+15] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG Reconstruction from Unstructured Programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

[Bit20] Bitdefender. Bitdefender. `http://www.bitdefender.com/`, June 2020.

[BJW+10] 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 *IEEE Symposium on Reliable Distributed Systems*, 2010.

[Bon94] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, 1994.

[Bov14a] Daniel Pierre Bovet. Implementing Virtual System Calls. `https://lwn.net/Articles/615809/`, October 2014.

[Bov14b] Daniel Pierre Bovet. Implementing Virtual System Calls. `https://lwn.net/Articles/615809/`, October 2014.

[Bro09] Jon Brodkin. With long history of virtualization behind it, IBM looks to the Future. `https://www.networkworld.com/article/2254433/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html`, April 2009.

[Bro14] Neil Brown. Control Groups Series by Neil Brown. `https://lwn.net/Articles/604609/`, July 2014.

[BVN16] Robert Buhren, Julian Vetter, and Jan Nordholz. The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture. In *International Conference on Information and Communications Security (ICICS)*. 2016.

[BYMX+06] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert Van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Ottawa Linux Symposium*, 2006.

[CAGN17] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[CAM+08] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[CBP+15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*, 2015.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

[CC00] Patrick Cousot and Radhia Cousot. Temporal Abstract Interpretation. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2000.

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[Chi08] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Pearson Education, 2008.

[CLH+15] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[CN01] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[Cor04] Jonathan Corbet. x86 NX Support. `https://lwn.net/Articles/87814/`, June 2004.

[Cor06] Corbet. A Bid to Resurrect Linux Capabilities. `https://lwn.net/Articles/199004/`, September 2006.

[Cor09] Jonathan Corbet. Seccomp and Sandboxing. https://lwn.net/Articles/332974/, May 2009.

[Cor10] Jonathan Corbet. The Kernel and the C Library as a Single Project. https://lwn.net/Articles/417647/, November 2010.

[Cor12a] Jonathan Corbet. Relocating RCU Callbacks. https://lwn.net/Articles/522262/, October 2012.

[Cor12b] Jonathan Corbet. Supervisor Mode Access Prevention. https://lwn.net/Articles/517475/, October 2012.

[Cor12c] Jonathan Corbet. Yet Another New Approach to Seccomp. https://lwn.net/Articles/475043/, January 2012.

[Cor15] Jonathan Corbet. Memory Protection Keys. https://lwn.net/Articles/643797/, May 2015.

[Cor16] Jonathan Corbet. Kernel Support for Miscellaneous (Your Favourite) Binary Formats v1.1. https://lwn.net/Articles/679310/, March 2016.

[Cor17] Jonathan Corbet. The Current State of Kernel Page-Table Isolation. https://lwn.net/Articles/741878/, December 2017.

[Cor18] Jonathan Corbet. Kernel Support for Control-Flow Enforcement. https://lwn.net/Articles/758245/, June 2018.

[Cor19] Jonathan Corbet. vDSO, 32-Bit Time, and Seccomp. https://lwn.net/Articles/615809/, August 2019.

[Cor20a] Jonathan Corbet. Conventions for Extensible System Calls. https://lwn.net/Articles/830666/, September 2020.

[Cor20b] Intel Corporation. Deep Dive: Intel Analysis of L1 Terminal Fault. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, June 2020.

[CP17] Scott A Carr and Mathias Payer. Datashield: Configurable Data Confidentiality and Integrity. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[CXS+05] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.

[CZM+14] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attack. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[CZW+17] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[Dat16] National Vulnerability Database. CVE-2016-8655. `https://nvd.nist.gov/vuln/detail/CVE-2016-8655`, August 2016.

[Dat17] National Vulnerability Database. CVE-2017-7308. `https://nvd.nist.gov/vuln/detail/CVE-2017-7308`, March 2017.

[DCCC08] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware Supported Rootkit Concealment. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[DGLS17] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[DKD+15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *ACM SIGPLAN Notices*, 2015.

[dml15] Xen devel mailing list. Alternate `p2m` design specification. `https://lists.xenproject.org/archives/html/xen-devel/2015-06/msg01319.html`, June 2015.

[DN14] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[Doc20a] Docker Inc. Docker. `https://www.docker.com/`, June 2020.

[Doc20b] Docker Inc. Docker Hub. `https://hub.docker.com/search?q=&type=image`, June 2020.

[Dom17] Domas, Christopher. Breaking the x86 ISA. *Black Hat, USA*, 2017.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[DWKJ+20] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. `sysfilter`: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[DZX13] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[Edg13] Jake Edge. Kernel Address Space Layout Randomization. `https://lwn.net/Articles/569635/`, October 2013.

[Edg15] Jake Edge. Inheriting Capabilities. `https://lwn.net/Articles/632520/`, February 2015.

[ELO+15] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[Eng20] Alexis Engelke. Lightweight Disassembler. `https://github.com/aengelke/lwdpy`, June 2020.

[Fer07] Ferrie, Peter. Attacks on More Virtual Machine Emulators. *Symantec Technology Exchange*, 2007.

[Fer19] Simon Ferquel. Docker & WSL 2 – The Future of Docker Desktop for Windows. `https://www.docker.com/blog/docker-hearts-wsl-2/`, June 2019.

[FGH+04] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2004.

[Fir20] FireEye Inc. FireEye. `https://www.fireeye.com/`, June 2020.

[Fle17] Matt Fleming. A Thorough Introduction to eBPF. `https://lwn.net/Articles/740157/`, December 2017.

[FLPV15] Emmanuel Fleury, Olivier Ly, Gérald Point, and Aymeric Vincent. Insight: An Open Binary Analysis Framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.

[Fou15a] Linux Foundation. Linux Standard Base Core Specification v5.0. *Whitepaper*, June 2015.

[Fou15b] The Linux Foundation. Open Container Initiative. `https://www.opencontainers.org/`, June 2015.

[Fou20] The Linux Foundation. containerd: An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. `https://containerd.io/`, June 2020.

[GA07] Andreas Gruenbacher and Seth Arnold. AppArmor Technical Documentation. `http://lkml.iu.edu/hypermail/linux/kernel/0706.1/0805/techdoc.pdf`, April 2007.

[GAWF07] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[GDXJ11] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *International Symposium on Reliable Distributed Systems (SRDS)*, 2011.

[GEN15] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[GKK+18] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[GKP+15] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information Flow Analysis of Android Applications in Droidsafe. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[GLS+17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*, 2017.

[Goo20a] Google. lmctfy – Let Me Contain That For You. `https://github.com/google/lmctfy`, June 2020.

[Goo20b] Google. The Chromium Projects. `https://www.chromium.org/developers/testing/control-flow-integrity`, June 2020.

[GPBP20] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[GPMP20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium*, 2020.

[GPZ23] Charlie Groh, Sergej Proskurin, and Apostolis Zarras. Free Willy: Prune System Calls to Enhance Software Security. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2023. To appear.

[GR03] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2003.

[Gra16] Aaron Grattafiori. Understanding and Hardening Linux Containers. *NCC Group Whitepaper*, June 2016.

[Gro18] Charlie Groh. Enhancing Security of Modern Linux Containers. Bachelor's Thesis, Technical University of Munich, 2018.

[Gro20] The PostgreSQL Global Development Group. PostgreSQL 9.3.25 Documentation: Pgbench. `https://www.postgresql.org/docs/9.3/pgbench.html`, June 2020.

[GTPJ16] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[GVJ14] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *IEEE Workshop on Mobile Security Technologies (MoST)*, 2014.

[GWT+96] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *USENIX Security Symposium*, 1996.

[HB17] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *International Conference on Virtual Execution Environments*, 2017.

[HCA+15] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *USENIX Security Symposium*, 2015.

[HDX+18] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. In *USENIX Annual Technical Conference*, 2018.

[HSA+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[IAJP18] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[Inc17] Red Hat Inc. CVE-2017-5123. `https://access.redhat.com/security/cve/cve-2017-5123`, October 2017.

[Inc20] RedHat Inc. rkt: A Security-minded, Standards-based Container Engine. `https://coreos.com/rkt/`, June 2020.

[Int19] Intel Corporation. *Intel Virtualization Technology for Directed I/O*. 2019.

[Int20a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. 2020.

[Int20b] Intel Corporation. Intel Architecture Instruction Set Extensions and Future Features. *Programming Reference*, 2020.

[Int20c] Intel Corporation. Intel Virtualization Technology for Connectivity. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/virtualization-technology-connectivity-brief.pdf, June 2020.

[Int21] Intel Corporation. Intel Trust Domain Extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419, August 2021.

[JBZ+14] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[JMG+02] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, 2002.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM—Software Protection for the Masses. In *IEEE Workshop on Software Protection*, 2015.

[JvSRZ16] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining Sandboxes. In *International Conference on Software Engineering (ICSE)*, 2016.

[KCB+17] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[KCL+18] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-Assisted Code Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[KCW+06] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing Malware With Virtual Machines. In *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[Ker13a] Michael Kerrisk. A Bid to Resurrect Linux Capabilities. https://lwn.net/Articles/534682/, January 2013.

[Ker13b] Michael Kerrisk. Namespaces in Operation, Part 1: Namespaces Overview. https://lwn.net/Articles/531114/, January 2013.

[KGP19] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *European Workshop on System Security (EuroSec)*, 2019.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[Kil73] Gary A Kildall. A Unified Approach to Global Program Optimization. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1973.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The Linux Virtual Machine Monitor. In *USENIX Security Symposium*, 2007.

[KKP03] Gaurav S Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[KPK12] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *USENIX Security Symposium*, 2012.

[KPK14] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium*, 2014.

[KSP+14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[KV08] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *International Conference on Computer-Aided Verification (CAV)*, 2008.

[KV10] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2010.

[KVL+14] Thomas Kittel, Sebastian Vogl, Tamas K. Lengyel, Jonas Pfoh, and Claudia Eckert. Code Validation for Modern OS Kernels. In *Workshop on Malware Memory Forensics (MMF)*, 2014.

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2009.

[Lab20] Redis Labs. How Fast Is Redis? `https://redis.io/topics/benchmarks`, June 2020.

[Lar14] Larsen, Per and Homescu, Andrei and Brunthaler, Stefan and Franz, Michael. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[LBL+18] Tien-Duy B Le, Lingfeng Bao, David Lo, Debin Gao, and Li Li. Towards Mining Comprehensive Android Sandboxes. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2018.

[LCLW14] 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*. Wiley Publishing, 2014.

[Len15] Tamas K Lengyel. *Malware Collection and Analysis via Hardware Virtualization*. PhD thesis, University of Connecticut, 2015.

[Len16] Tamas K Lengyel. Stealthy Monitoring With Xen Altp2m. `https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m`, April 2016.

[LHKS15] JungSeung Lee, HyoungMin Ham, InHwan Kim, and JooSeok Song. POSTER: Page Table Manipulation Attack. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[Lia09] Siarhei Liakh. NX Protection for Kernel Data. `https://lwn.net/Articles/342266/`, July 2009.

[Lib19] Libsodium. Libsodium Documentation. `https://libsodium.gitbook.io/doc/memory_management`, April 2019.

[Lib20] LibVMI. LibVMI Virtual Machine Introspection. `http://libvmi.com`, June 2020.

[Lin20a] Linux Foundation. Xen Project. `https://www.xenproject.org/`, June 2020.

[Lin20b] Linux Kernel Virtual Machine. Kernel Virtual Machine. `http://www.linux-kvm.org`, June 2020.

[Lit08] Litty, Lionel and Lagar-Cavilla, H. Andrés and Lie, David. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium*, 2008.

[LKE15] Tamas K. Lengyel, Thomas Kittel, and Claudia Eckert. Virtual Machine Introspection with Xen on ARM. In *Workshop on Security in highly connected IT systems (SHCIS)*, 2015.

[LKPE14] Tamas K Lengyel, Thomas Kittel, Jonas Pfoh, and Claudia Eckert. Multi-Tiered Security Architecture for ARM via the Virtualization and Security Extensions. In *International Workshop on Database and Expert Systems Applications (DEXA)*, 2014.

[LLV20] LLVM. Control Flow Integrity. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`, June 2020.

[LLW+18] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[LMP+14] Tamas K Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[LNW+19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium*, 2019.

[LS01] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, 2001.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[LSK18] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[Ltd20] Canonical Ltd. Infrastructure for Container Projects – LXC. `https://linuxcontainers.org/lxc/introduction/`, June 2020.

[LYGC17] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[LZC+15] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[MANP17] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[MBBM15] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[McK07] Paul McKenney. What is RCU, Fundamentally? `https://lwn.net/Articles/262464/`, December 2007.

[MFPC10] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2010.

[MHHW18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *European Workshop on System Security (EuroSec)*, 2018.

[MHJM13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v Application Binary Interface AMD64 Architecture Processor Supplement. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[Mic11] Microsoft. The Drawbridge Project. `https://www.microsoft.com/en-us/research/project/drawbridge/`, September 2011.

[Mic16a] Microsoft. Pico Process Overview. https://docs.microsoft.com/en-us/archive/blogs/wsl/pico-process-overview, May 2016.

[Mic16b] Microsoft. Windows Subsystem for Linux Overview. https://docs.microsoft.com/en-us/archive/blogs/wsl/windows-subsystem-for-linux-overview, April 2016.

[Mic17a] Microsoft. Documentation: Virtualization-based Security (VBS). https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs, May 2017.

[Mic17b] Microsoft. Introducing the Host Compute Service (HCS). https://docs.microsoft.com/en-us/virtualization/community/team-blog/2017/20170127-introducing-the-host-compute-service-hcs, January 2017.

[Mic20a] Trend Micro. XORDDoS, Kaiji Botnet Malware Variants Target Exposed Docker Servers. https://blog.trendmicro.com/trendlabs-security-intelligence/xorddos-kaiji-botnet-malware-variants-target-exposed-docker-servers/, June 2020.

[Mic20b] Microsoft. Control Flow Guard. https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard, June 2020.

[Mic20c] Microsoft Security. Introducing Kernel Data Protection, a New Platform Security Technology for Preventing Data Corruption. *Microsoft Security Blog*, 2020.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *USENIX winter*, 1993.

[MJ98] David Mosberger and Tai Jin. Httperf — A Tool for Measuring Web Server Performance. In *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[Mob20] Moby. Docker Source Code: Default Seccomp Profile. https://github.com/moby/moby/blob/v17.05.0-ce/profiles/seccomp/default.json, June 2020.

[Mom20] Marius Momeu. Hardening the Linux Kernel Slab Allocator. Master's thesis, Technical University of Munich, 2020.

[MPR+21] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. SEVerity: Code Injection Attacks Against Encrypted Virtual Machines. In *IEEE Workshop on Offensive Technologies (WOOT)*, 2021.

[MSKH02] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, 2002.

[MvdKG22] Alyssa Milburn, Erik van der Kouwe, and Cristiano Giuffrida. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[MWK+18] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[NMW02] George C Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[NZ17] Zhenyu Ning and Fengwei Zhang. Ninja: Towards Transparent Tracing and Debugging on Arm. In *USENIX Security Symposium*, 2017.

[NZMZ09] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *ACM SIGPLAN Notices*, 2009.

[NZMZ10] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. *ACM SIGPLAN Notices*, 2010.

[Ope19a] Open Web Application Security Project (OWASP). C-Based Toolchain Hardening. `https://www.owasp.org/index.php/C-Based_Toolchain_Hardening`, January 2019.

[Ope19b] OpenSSL. OpenSSL Manpages v1.0.2. `https://www.openssl.org/docs/man1.0.2/man3/bn.html`, April 2019.

[Ora20] Oracle. Oracle Solaris Zones. `https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm`, June 2020.

[PaX03] PaX Team. Address Space Layout Randomization (ASLR). `https://pax.grsecurity.net/docs/aslr.txt`, March 2003.

[PaX20] PaX Project. Pageexec. `http://pax.grsecurity.net/docs/pageexec.txt`, June 2020.

[Pay12] Bryan D. Payne. Simplifying Virtual Machine Introspection Using LibVMI. *Sandia Report*, 2012.

[PCSL08] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[Pfo13] Jonas Pfoh. *Leveraging Derivative Virtual Machine Introspection Methods for Security Applications*. PhD thesis, Technical University of Munich, 2013.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 1974.

[PKE15] Sergej Proskurin, Fatih Kilic, and Claudia Eckert. Retrospective Protection utilizing Binary Rewriting . In *Deutscher IT-Sicherheitskongress*, 2015.

[PKZ18] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection. In *IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, 2018.

[PLM+18] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[PMG+20] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[PMK12] Sergej Proskurin, David McMeekin, and Achim Karduck. Smart Camp: Building Scalable and Highly Available IT-Infrastructures. In *IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, 2012.

[Por05] Chris Porthouse. High Performance Java on Embedded Devices – Jazelle DBX Technology: ARM Acceleration Technology for the Java Platform. *Whitepaper*, 2005.

[PPK+17] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kR^X: Comprehensive Kernel Protection Against Just-In-Time Code Reuse. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[Pro03] Niels Provos. Improving Host Security With System Call Policies. In *USENIX Security Symposium*, 2003.

[Pro16] Sergej Proskurin. Forensic Analysis Utilizing Virtualization On-the-Fly. Master's thesis, Technical University of Munich, 2016.

[PSE09] Jonas Pfoh, Cristian Schneider, and Claudia Eckert. A Formal Model for Virtual Machine Introspection. In *Workshop on Virtual Machine Security (VMSec)*, 2009.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-Based System Call Tracing for Virtual Machines. In *International Workshop on Advances in Information and Computer Security (IWSEC)*, 2011.

[PWS15] Sergej Proskurin, Michael Weiss, and Georg Sigl. seTPM: Towards Flexible Trusted Computing on Mobile Devices based on GlobalPlatform Secure Elements. In *International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 2015.

[QK19] Rian Quin and Brendan Kerrigan. Bareflank Hypervisor. `https://bareflank.github.io/hypervisor/`, September 2019.

[QPY18] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software Through Piece-Wise Compilation and Loading. In *USENIX Security Symposium*, 2018.

[Qua17] Qualcomm. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, January 2017.

[Rad20] Radare2. Libre and Portable Reverse Engineering Framework. https://rada.re/n/, June 2020.

[RDDC+17] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[Rek20] Rekall Forensics. Advanced Forensic and Incident Response Framework. http://www.rekall-forensic.com/, June 2020.

[Res20] Microsoft Research. Z3 Documentation. https://github.com/Z3Prover/z3/wiki/Documentation, June 2020.

[Rio20] Matteo Riondato. BSD Jails. https://www.freebsd.org/doc/handbook/jails.html, June 2020.

[RNMJ17] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New Directions for Container Debloating. In *Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[RT07] Joanna Rutkowska and Alexander Tereshkin. IsGameOver () Anyone. *Black Hat, USA*, 2007.

[Rus08] Rusty Russell. Virtio: Towards a De-Facto Standard for Virtual I/O Devices. In *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[Rut04] Joanna Rutkowska. Red Pill... Or How to Detect VMM Using (Almost) One CPU Instruction. https://securiteam.com/securityreviews/6z00h20bqs/, November 2004.

[Rut06a] Joanna Rutkowska. Introducing Stealth Malware Taxonomy. In *COSEINC Advanced Malware Labs*, 2006.

[Rut06b] Joanna Rutkowska. Subverting VistaTM Kernel for Fun and Profit. *Black Hat, USA*, 2006.

[SA.20] Hex-Rays SA. IDA Pro. https://www.hex-rays.com/products/ida/, June 2020.

[Sal17] Chris Salls. Exploiting CVE-2017-5123 With Full Protections. SMEP, SMAP, and the Chrome Sandbox! https://salls.github.io/Linux-Kernel-CVE-2017-5123, November 2017.

[SAM14] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. Cardinal Pill Testing of System Virtual Machines. In *USENIX Security Symposium*, 2014.

[SB05] Sherri Sparks and Jamie Butler. Shadow Walker: Raising the Bar for Rootkit Detection. *Black Hat, Japan*, 2005.

[SCL+18] Bin Shi, Lei Cui, Bo Li, Xudong Liu, Zhiyu Hao, and Haiying Shen. Shadow-Monitor: An Effective In-VM Monitoring Framework with Hardware-Enforced Isolation. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[SFV20] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. `https://www.kernel.org/doc/html/latest/security/lsm.html`, June 2020.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[SMD+13] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[SMJ+17] Matt Shockley, Chris Maixner, Ryan Johnson, Mitch DeRidder, and W Michael Petullo. Using VisorFlow to Control Information Flow without Modifying the Operating System Kernel or its Userspace. In *International Workshop on Managing Insider Security Threats (MIST)*, 2017.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., 2005.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM European Conference on Computer Systems (EuroSys)*, 2007.

[SS72] Michael D Schroeder and Jerome H Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 1972.

[Sti20] Stichting Cuckoo Foundation. Cuckoo: Automated Malware Analysis. `https://www.cuckoosandbox.org/`, June 2020.

[STL+15] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the

Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[SWH+15] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice — Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State Of) the Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[SXC17] Bing Sun, Chong Xu, and Stanley Chong. The Power of Data-Oriented Attacks. *Black Hat, Asia*, 2017.

[Syn14] Synopsys. The Heartbleed Bug. `http://heartbleed.com/`, April 2014.

[TBZ+18] Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K Lengyel, and Kyle J Temkin. Who Watches the Watcher? Detecting Hypervisor Introspection from Unprivileged Guests. 2018.

[The20a] The Linux Kernel Archieves. Ftrace – Function Tracer. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`, June 2020.

[The20b] The New York Times. Japanese Supercomputer Is Crowned World's Speediest. `https://www.nytimes.com/2020/06/22/technology/japanese-supercomputer-fugaku-tops-american-chinese-machines.html`, June 2020.

[TKFC15] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[Tor14] Jacob Torrey. MoRE Shadow Walker: TLB-splitting on Modern X86. *Black Hat, USA*, 2014.

[TRC+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[UPBSB15] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[vdV06] Arjan van de Ven. Add `-fstack-protector` Support to the Kernel. `https://lwn.net/Articles/193307/`, July 2006.

[VKSE13] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. X-Tier: Kernel Module Injection. In *International Conference on Network and System Security (NSS)*, 2013.

[VMR20] VMRay. VMRay: X-Ray Vision for Malware. `https://www.vmray.com`, June 2020.

[Vog15] Sebastian Vogl. *Data-Only Malware*. PhD thesis, Technical University of Munich, 2015.

[Vol20] Volatility Foundation. The Volatility framework. `http://www.volatilityfoundation.org/`, June 2020.

[Wal02] Carl A Waldspurger. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review (OSR)*, 2002.

[WBD+16] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[WBL+19] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based Side Channels Through Secret-Augmented Abstract Interpretation. In *USENIX Security Symposium*, 2019.

[WD01] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2001.

[Win20] WineHQ. Wine is not an Emulator. `https://www.winehq.org/`, June 2020.

[WKKWK+20] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[WLX+17] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[WMA+19] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2019.

[WVBM+18] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018.

[WvOS05] Glenn Wurster, Paul C van Oorschot, and Anil Somayaji. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *IEEE Symposium on Security and Privacy (S&P)*, 2005.

[WWZ+19] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization. In *USENIX Security Symposium*, 2019.

[WZH+11] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, 2011.

[Xen16a] Xen Project. Xen Security Advisory 203. `https://xenbits.xen.org/xsa/advisory-203.html`, December 2016.

[Xen16b] Xen Project. Xen Security Advisory 204. `https://xenbits.xen.org/xsa/advisory-204.html`, December 2016.

[YIRS17] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals, Seventh Edition, Part 1*. Microsoft Press, 2017.

[YM87] William D Young and John McHugh. Coding for a Believable Specification to Implementation Mapping. In *IEEE Symposium on Security and Privacy (S&P)*, 1987.

[Yu11] Fenghua Yu. Enable/Disable Supervisor Mode Execution Protection. `https://goo.gl/utKHno`, May 2011.

[YY12] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium*, 2012.

[Zab18] Adam Zabrocki. Linux Kernel Runtime Guard (LKRG) under the Hood. *CONFidence*, 2018.

[ZC11] Matteo Zanioli and Agostino Cortesi. Information Leakage Analysis by Abstract Interpretation. In *International Conference on Current Trends in Theory and Practice of Computer Science*, 2011.

[ZH18] Philipp Zieris and Julian Horsch. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[ZLS+15] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using Hardware Features for Increased Debugging Transparency. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[ZLSS13] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[Zov06] Dino A. Dai Zovi. Hardware Virtualization Rootkits. *Black Hat, USA*, 2006.

[ZSS+16] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.