Strengthening System Security on the ARMv7 Processor Architecture with Hypervisor-based Security Mechanisms

vorgelegt von
Julian Vetter (M.Sc.)
geb. in Lindenfels

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende:	Prof. Dr. Anja Feldmann, Technische Universität Berlin
Gutachter:	Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin
Gutachter:	Prof. Dr. Marian Margraf, Freie Universität Berlin
Gutachter:	Prof. Dr. Shay Gueron, University of Haifa

Tag der wissenschaftlichen Aussprache: 19.05.2017

Berlin 2017

Publications related to this Thesis

The work presented in this thesis resulted in the following peer-reviewed publications:

- *The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture*, Robert Buhren, **Julian Vetter**, Jan Nordholz, 18th International Conference on Information and Communications Security (ICICS), Singapore, November 29th, 2016
- *Uncloaking Rootkits on Mobile Devices with a Hypervisor-Based Detector*, **Julian Vetter**, Matthias Petschick-Junker, Jan Nordholz, Michael Peter, Janis Danisevskis, 18th International Conference on Information Security and Cryptology (ICISC), Seoul, South Korea, November 25-27th, 2015
- *XNPro: Low-Impact Hypervisor-Based Execution Prevention on ARM*, Jan Nordholz, **Julian Vetter**, Michael Peter, Matthias Petschick and Janis Danisevskis, 5th International Workshop on Trustworthy Embedded Devices (CCS TrustED), Colorado, USA, October 12-16th, 2015
- *Undermining Isolation through Covert Channels in the Fiasco.OC Microkernel*, Michael Peter, Matthias Petschick, **Julian Vetter**, Jan Nordholz, Janis Danisevskis, Jean-Pierre Seifert, 30th International Symposium on Computer and Information Sciences (ISCIS), London, UK, September 21-25th, 2015

Additionally, Julian Vetter has authored the following publications:

- *Fault Attacks on Encrypted General Purpose Compute Platforms*, Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, **Julian Vetter**, 7th Conference on Data Application Security and Privacy (CODASPY), Scottsdale, USA, March 22-24th, 2017
- *Graphical User Interface for Virtualized Mobile Handsets*, Janis Danisevskis, Michael Peter, Jan Nordholz, Matthias Petschick, **Julian Vetter**, 4th International Workshop on Mobile Security Technologies (S&P MoST), San Jose, CA, May 21st, 2015

Abstract

The computing landscape has significantly changed over the last decades. The devices we use today to interact with digital content have shifted away from stationary computers towards ubiquitous network-connected devices such as mobile phones. The success was mainly driven by two trends: the fast evolution of communication and computing hardware and a rapid change of the system software away from proprietary special purpose operating systems towards open commodity operating systems. However, this mobile trend also attracted adversaries.

In this thesis, we therefore raise the question whether commodity operating systems are suitable to protect users of such devices from common attacks; e.g., malware or rootkits). Arguably, commodity operating systems such as Linux provide an appealing option because of their extensive hardware device support and their broad selection of applications. However, they are built around a monolithic kernel and if a highly privileged component is breached, an adversary can take over the entire system. This renders them unsuitable for applications that have higher security demands.

In response, researchers explored several approaches to gain the desired security properties. Two prime examples are hypervisors and microkernels, both of which promise a higher degree of isolation between components than is provided by commodity operating systems. While most hypervisors also go for a monolithic kernel design, they achieve the better isolation capabilities through a reduced functionality, which in turn leads to less complex interfaces and a way smaller trusted computing base. Microkernel-based systems, on the other, hand achieve their security properties by putting kernel-level functionality into user processes, thereby also reducing their trusted computing base, while still providing a similar functionality as a general purpose operating system such as Linux. Both seem like promising options to protect the users from attacks. However, as we show in this thesis, both paradigms have issues if deployed carelessly.

To assure a minimal performance impact when running a hypervisors, hardware vendors added hardware virtualization extensions to their processors. However, if the access to these extensions is not properly controlled, they pose a severe security threat and can be used to subvert the operating system. We show how the virtualization extensions can be leveraged to take over the highly privileged hypervisor mode on ARMv7 based devices. Subsequently, we plant into the said

mode a rootkit, which is very hard to spot and to remove.

As an alternative to a hypervisor-based system architecture, we investigate a widely used microkernel with respect to the isolation capability it promises. We assess Fiasco.OC, a microkernel that claims to be suitable for the construction of highly compartmentalized systems. But even this seemingly secure system software cannot uphold the promised isolation capabilities. In several scenarios, we show how to create high-bandwidth covert channels between two user-level entities, undermining all isolation efforts.

Based on the outcome of our audit, we propose a system architecture for security-critical devices, based on a small statically partitioned Type-I hypervisor. The only aspect that speaks against a hypervisor-based design is the coarse-grained isolation hypervisors in general provide. We bridge this gap with two security mechanisms embedded into the hypervisor that narrow done the attack surface inside individual guests. The first mechanism enforces strict memory attributes to prevent common code reuse attacks. The second mechanism allows us to take snapshots of the memory of a guest, which is a powerful way to detect the presence of rootkits, which are, by their nature, otherwise hard to locate.

Zusammenfassung

In den letzten Jahren hat sich die IT-Landschaft drastisch verändert. Um mit digitalen Inhalte zu interagieren verwenden wir kaum noch stationäre Computer, stattdessen verwenden wir ubiquitäre vernetze Geräte (z.B. Smartphones). Neben einer rapiden Evolution der Hardware, gab es auch drastische Änderungen der Sotftware. Wo vor einigen Jahren noch proprietäre Lösungen zum Einsatz kamen, heißt es heute offene Standardbetriebssysteme. Aber gerade die Tatsache das diese offen und standardisiert sind, heißt auch das sie eben genauso anfällig sind wie bisher nur Desktop- und Serversysteme.
In dieser Thesis evaluieren wir daher, ob solche Betriebssysteme geeignet sind, die Nutzer auch in diesen neuen Domänen vor Bedrohungen wie Malware und Rootkits zu schützen. Linux stellt natürlich eine attraktive Lösung dar. Es verfügt über exzellente Hardwareunterstüzung und eine große Auswahl an Applikationen. Allerdings basieren diese Standard-Betriebssysteme auf einem monolithischen Kern. Wenn also eine hoch privilegierte Komponente von einem Angreifer übernommen wird, ist automatisch das gesamte System kompromittiert. Durch diese Eigenschaft sind sie ungeeignet für Systeme die ein höheres Maß an Sicherheit fordern.
Forscher haben sich daher nach anderen Ansätzen umgesehen, um die gewünschten Isolationseigenschaften zu erhalten. Zwei bekannte Beispiele hierbei sind Hypervisor und Microkerne, beide versprechen einen höheren Grad an Isolation zwischen Systemkomponenten. Während viele Hypervisor auch über einen monolithische Kern verfügen, erreichen sie doch eine bessere Isolation durch eine reduzierte Anzahl an Funktionen. Diese geringere Komplexität führt zu weniger komplexen Schnittstellen und eine reduzierte Trusted Computing Base. Microkern-basierte System erreichen ihre Sicherheitseigenschaften dadurch, dass Kernkomponenten in Nutzerprozesse ausgelagert werden. Dadurch erreichen sie auch eine reduzierte Trusted Computing Base und haben doch eine ähnlich vielfältige Funktionalität wie andere Standardbetriebssysteme wie z.B. Linux. Beide scheinen eine gute Option zu sein um Nutzer vor Angriffen zu schützen. Allerdings zeigen wir in dieser Thesis, dass beide Systeme Probleme aufweisen wenn sie unvorsichtig eingesetzt werden. Um nur minimale Leistungseinbußen beim Einsatz von Hypervisor-basierten Systemen zu haben, stellen Hardware Hersteller Virtualisierungserweiterungen für ihre Prozessoren zu Verfügung. Wenn der Zugriff auf diese Erweiterungen allerdings nicht korrekt behandelt wird kann dies fatale Folgen für die Sicherheit des Sys-

tems haben. Wir zeigen auf einer ARMv7-basierten Plattform, dass ein Angreifer Kontrolle über die Virtualisierungserweiterungen erlangen kann um ein Rootkit im hoch privilegierten Hypervisor-Modus zu plazieren. Für das Betriebssystem ist das Aufspüren oder Entfernen eines solchen Rootkits extrem schwierig.
Darüber hinaus, evaluieren wir Fiasco.OC, ein Microkern, welcher einzelne Nutzerkomponenten stärker von einander isoliert. Aber auch dessen Architektur zeigt Schwächen und erlaubt uns Daten zwischen zwei Nutzerkomponenten auszutauschen, die eigentlich durch keinen Kommunikationskanal verbunden sind.
Basierend auf dem Ergebnis unserer Evaluation stellen wir eine neue Systemarchitektur vor, welche auf einem kleinen statisch partitionierten Type-I Hypervisors beruht. Da Hypervisor im allgemeinen nur eine grobe Unterteilung von Softwarekomponenten ermöglichen, stellen wir zwei Sicherheitsmechanismen vor, welche in den Hypervisor eingebettet sind, um diese Lücke zu schließen. Diese sollen mögliche Angriffsvektoren auf das Gastbetriebssystem minimieren. Der erste Mechanismus stellt bestimmte Speicherattribute sicher, um bekannte “Code Reuse” Angriffe zu verhindern. Der zweite Mechanismus ermöglicht das Erstellen von Speicherabbildern von Gastspeicher. Ein Satz von Nutzerapplikationen ermöglicht uns dann Kerndatenstrukturen zu rekonstruieren um Rootkits aufzudecken.

Contents

Part I

Preliminaries & Assumptions

1 Introduction

With the introduction of the first iPhone in 2007, the trend of mobile computing took its course. The decreasing manufacturing costs and the advances in computing and communication hardware led to entirely new ways of how we use computers. Especially ARM-based devices have seen exponential growth since virtually all mobile systems today are equipped with an ARM-based SoC (System-on-Chip). But this new form of computing also posed additional requirements on the existing software stack. To accommodate this increased complexity, the OSs (Operating Systems) had to evolve. Instead of running a specialized OS, today the majority of mobile devices run one of two commodity OSs (Google's Android or Apple's iOS). For this discussion we focus on Android, because it is more accessible in terms of licensing and platform support. Still, no matter which of the two we consider they share common ground in giving the users the ability to install additional applications and both support a wide range of connectivity options (e.g. cellular, WLAN, Bluetooth, etc.). The downside of this trend is that radical changes of that magnitude involve many security risks. Commodity OSs are complex, provide an enlarged attack surface and the ubiquitous network connectivity exposes the respective systems to remote adversaries [110, 37].

Mobile devices nowadays are highly personalized. Users store sensitive data on them and perform sensitive tasks, e.g., online banking. Still, many users are either uneducated or just careless in operating it, e.g., by installing applications from untrustworthy sources or by not checking the requested permissions of installed applications. Adversaries quickly adapted to this behaviour and started repackaging existing applications with malware and uploading them to various alternative stores [75]. But not only alternative stores host malware, adversaries also managed to trick Google's safeguard [84] and deployed malware in the official store. It is important to note that even though many users operate their mobile device carelessly, it is also extremely difficult to decide whether an application is benign or not (e.g., based on the permissions it requests). Moreover, Android only allows for very coarse grained assignment of permissions, e.g., access to entire calendar, and various researchers showed how to circumvent the permission system [2, 43].
But adversaries did not limit their efforts to unsophisticated application-based malware. Recent incidents revealed that high value targets, e.g, government employees, were victims of sophisticated attacks targeting their mobile devices [63, 47, 51].

Adversaries used complex rootkits, which were carefully constructed to not be easily detectable by application-based anti-malware solutions.

Even though researchers tried to address these issues, already on the Desktop before, e.g., by implementing more restrict access control mechanisms [16, 114]. These efforts not only proved themself to be difficult due to the monolithic nature of commodity OS, but also do not prevent sophisticated adversaries from installing deeply embedded malware (e.g. rootkits) on devices. Thus, a completely revised system architecture might be favorable.
For several years now virtualization is the most common paradigm on server systems. It provides two main advantages: first, achieving a better resource utilization of a physical machine and second, a stronger isolation between software components and therefore preventing a full system breach if a single component is compromised. However, in order to implement the virtualization paradigm efficiently, hardware support is necessary. For the processor architecture dominant in the server domain (x86), vendors released hardware extensions for their respective processors [108, 96] in 2004.
Researchers then explored the paradigm for a range of other domains [97, 59, 76, 15], mainly for its second attribute – the stronger isolation properties. Among the proposed systems were also mobile devices. The adoption of virtualization for the mobile market at that time was however limited, mainly because mobile processors lacked hardware virtualization support and also because of the increased memory requirements when running two OS instead of one. Both aspects were problematic, because system integrators had to draw on para-virtualization, which required them to make changes to the OS and the memory needs could simple not be served by the mobile devices at the time.

But virtualization is not the only paradigm that promises a better isolation and an increased system security. In the past, researchers also studied systems with a very small TCB (Trusted Computing Base)[1] to decrease the attack surface and still provide strong isolation properties. These microkernels [45, 83] provide functionality similar to the one provided by commodity OS. They also provide a programming interface featuring processes and interprocess communication facilities. However, the microkernel paradigm can maintain a smaller TCB by demanding that the kernel to be free of policies. This is achieved by removing parts of the kernel code, e.g., drivers, memory management from the privileged domain and putting them into unprivileged user processes. Then, also bug-induced damage cannot affect the entire system.
The approach has long been acknowledged in academia; yet, its adoption was

[1] The TCB of a system encompasses all components (software and hardware) that are essential for its security. That is, if one component of the TCB contains a bug or is vulnerable to attacks the security of the entire system is at risk.

limited because porting applications to these new microkernel APIs takes time, and limited resources prevented a widespread adoption. Still, the microkernel paradigm found its niche. Notably, the L4 family of microkernels [73] held its ground and found appliance in some security systems. The commercial derivative OKL4 [57] is deployed on the secure enclave processor of the latest iPhone [61]. Fiasco.OC [46], another L4 derivative, attracted system designers because of its open source license and its ability to run VMs (Virtual Machine) alongside native microkernel processes. Hence, it found application in a secure mobile architecture [72], combining an encapsulated Android with native microkernel processes that provide access to security-critical components, e.g., a smartcard).
When taking a closer look at such a secure mobile architecture, it becomes clear that the ability to encapsulate Linux in form of Android is a key requirement. But running Linux on a microkernel is for several reasons ill-advised. Either, the Linux kernel has to be modified to run on the microkernel API [69] directly, but this not only exposes a complex interface to the hosted Linux but also requires a considerable porting effort for every new Linux kernel version. The other option is that the microkernel supports running VMs. This, however, requires to run a VMM (Virtual Machine Monitor) on top of the microkernel [72]. The advantage of this approach is that the VMM wraps the extensive microkernel API and hides it from guest systems. Still, the microkernel underneath provides a lot of functionality that is not required when only hosting VMs, making the TCB unnecessarily large.

In other domains such as avionics [89] and automotive [48], system architectures already feature small TCBs with statically assigned resources (policy free) and narrow interfaces. This effectively combines the two advantages of HVs (hypervisor) and microkernels – a small TCB and no policies in the kernel, as demanded by the microkernel paradigm, with a simple programming interface as provided by most HVs. However, the *separation kernel*, proposed by Rushby [93] in 1981, has clearly specified workloads, which make the implementation of such a design paradigm a rewarding task for system integrators.
The commodity software solutions today, regardless of their implemented design principle (microkernel, HV or general purpose OS), not only face entirely different challenges but also try to be as flexible as possible. In the desktop and server domain, systems have to be prepared for various types of workloads without prior knowledge about their resource utilization (memory and CPU time) and, of course, the solutions want to give the users the full freedom to execute additional processes or VMs on demand. In the mobile domain, on the other hand, it seems that all efforts so far to build a secure mobile architecture were either driven by principle instead of pragmatism or were simply introduced at the wrong time (and then lacking resources or hardware support).

1.1 Thesis Motivation

Now the question arises whether an architecture can be designed for mobile devices patterned after Rushby's seperation kernel. The foundation is there: ARM released its VE [79] in 2011, allowing for well performing virtualization on mobile devices. The goal is to achieve a higher degree of isolation, narrow and simple interfaces, a very small TCB without any policies inside the system software and still provide the needed flexibility to fulfill the requirements of the user.

1.1.1 Problem Overview

Given the described issues and taking the blueprint of Rushby's separation kernel, it is reasonable to assume that the concept is well applicable to today's mobile devices and might provide an appealing option. But first we have to take a look at the requirements that are imposed on today's mobile devices.

As already discussed, Linux (in form of Android) is the main OS in the mobile market because of its versatility (open source, software diversity, hardware support, etc.), so an essential capability of the envisioned architecture is to encapsulate it.
But unlike on a server, where the HV must spawn new VMs on demand (for load-balancing and overall hardware utilization), such a feature is barely useful on a mobile device. Only a predefined set of security-critical services will run on the device, thus the number of VMs is fixed. This will not change during runtime; nor will the user want to start additional ones. Of course, inside individual VMs that run Linux, the user is free to execute additional processes.
Moreover, hardware resources such as main memory can be statically assigned. Each VM get its fixed share of the memory, making any form of memory management policy in the HV obsolete. Because, again, it can be determined beforehand how much memory the security-critical service might need, the rest is then assigned to Linux. Figure 1.1 briefly depicts our envisioned secure system architecture, containing a VM hosting the rich OS (e.g. Linux) and an additional VM hosting a security critical service (might be hosted on Linux, but does not have to).

Such an architecture settles the issue with complex subsystems, confusing or complicated interfaces and shrinks the TCB. But, depending on the application the isolation granularity HVs in general provide, it might be too coarse. Unlike microkernels, which can isolate different processes, HVs only export a CPU-like interface, thus isolating at VM granularity. Commonly, the security inside individual VMs is left to security applications in the VM. But, of course, these applications, e.g.,

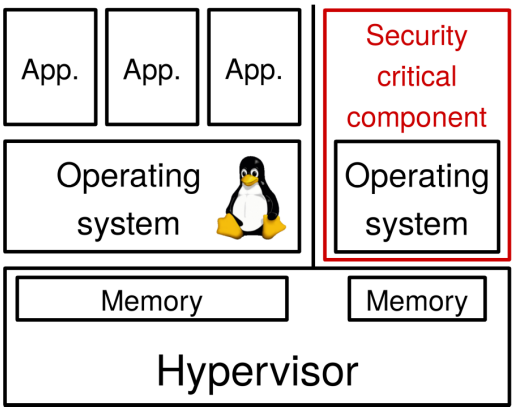


Fig. 1.1: Proposed security architecture based on a statically partitioned HV

virus scanner, IDS, firewall, etc., are not afforded any additional protection from an adversary inside the VM.

1.1.2 Thesis Statement

In this thesis, we investigate a security architecture suitable for mobile devices that leverages the isolation properties of a HV not only to separate different system components from each other. We also use the HV as a vehicle to implement defense mechanisms to increase the security inside individual VMs without giving a potential adversary inside a VM the chance to outright disable them.

Our design decisions are driven by the following principle. When integrating software components on a hardware platform, isolation should be a key attribute of the underlying system software. This allows not only to isolate security critical components from the rest of the system but also allows for an integration of defense mechanisms into the system architecture decoupled from the already complex and vulnerable rich OS to detect or even prevent intruders from attacking the said.
Therefore, in this thesis, we propose the following statement:

> *"To integrate several software components on a common [mobile] platform, system designers should utilize small statically partitioned HVs with well defined and narrow interfaces. Additionally, security features should be small, modular and transparent to the hosted [rich] operating systems."*

1.2 Thesis Contribution

In the previous section, we proposed a secure system architecture for mobile devices. But before we discuss the defense mechanisms we integrated into our hypervisor (in Part III), we want to substantiate the claim that commodity systems are indeed ill-suited to build secure system architectures (in Part II).
We examine how common OS expose interfaces to the virtualization extensions on ARM-based SoCs. We exploit these interfaces to gain higher privileges than the OS. We then place a rootkit in this highly privileged execution mode to spy on the OS. Furthermore, we explore vulnerabilities in an existing microkernel solution, which lead to a denial-of-service attack, and more severely, to high-bandwidth covert channels. Our findings allow us to undermine all efforts to isolate components in a system built on top of this kernel.

Our findings substantiate the previously proposed statement to carefully design narrow interfaces, and not build security- or safety-critical software based on complex commodity systems. As an alternative, we propose two security concepts integrated into a statically partitioned HV. Both security mechanism ought to bridge the gap between the relatively coarse-grained isolation properties provided by the HV and the process-based isolation provided by the OS running in the VM.
The first security mechanism enforces execution prevention capabilities to rule out common code reuse attacks. The second mechanism can uncover rootkits that might have infiltrated the kernel in a VM. Both mechanisms are designed for the ARMv7 processor architecture. The first mechanism is OS agnostic and can be configured to work with any OS, whereas the second mechanism is tailored towards threats imperiling Android systems. Both security mechanisms are part of a statically partitioned HV and are therefore out of the reach of an adversary who might have infiltrated the OS kernel in a VM.

1.3 Thesis Structure

This thesis is structured as follows. Part I, apart from this introduction, explains the technological background required to understand the concepts presented throughout this thesis. Specifically, in Chapter 2, we will cover the fundamentals of the ARMv7 processor architecture, the different processor modes, device handling and different processor extensions, along with a survey of related work in Chapter 3.
In Part II, we will substantiate our statement that security-critical systems should not be built on commodity system software by examining the security of two such systems. Specifically, in Section 4, we analyze how the Linux kernel handles the interface to ARM's hardware VE. With several attack vectors, we show that we can

take over the HV mode and install a rootkit into it, thereby getting full control over the OS on the device. In Section 5, we analyze the isolation properties of the Fiasco.OC microkernel. As a result, we can establish covert channels between two native L4 processes. This suggests that it cannot uphold the said properties.
We demonstrate the feasibility of the previously proposed security architecture in Part III. In Chapter 7, we present an execution prevention mechanisms that counters several code reuse attacks common in commodity OS. We show a HV-based rootkit detection solution in Chapter 6. Both are part of a small statically partitioned HV.
Finally, in Part IV we briefly conclude our research (Chapter 8) and provide directions for future research (Chapter 9), respectively.

2 ARM Processor Architecture

The following chapter will provide a brief background on the ARMv7 processor architecture [6]. This information by no means represents a complete overview of these topics. The specification for the ARMv7 processor architecture is however publicly available but comprises a large number of documents. Thus, we refer the interested reader to [7, 12, 9, 10]. We also specifically focus on the Cortex-A7 [27] and Cortex-A9 [28] processors, respectively.

It is important to understand that there is no single ARM system architecture. Because, unlike Intel or AMD who design and manufacture x86 processors, ARM only designs and sells IP (Intellectual Property) cores to other companies (e.g. Samsung, Allwinner, Qualcomm and Apple).
Furthermore, ARM's system architecture is built in a modular way. So, they not only design processor cores (e.g. Cortex-A7), but also peripheral components that are usually tightly integrated into the chip (e.g. UART, interrupt controller, graphics unit). But, companies that buy the license to manufacture an ARM processor are not obliged to also use these optional components. Instead, many companies, only license the ARM processor core and design other components on their own[1]. The resulting architecture is called an SoC. The SoC integrates multiple components along the processor, but depending on the SoC integrator/manufacturer the design significantly differs.

Moreover, ARM also sells "source" licenses of their IP cores. Effectively, allowing manufacturers that hold such a license to make changes to the processor's core design. Apple and Qualcomm are two prominent representatives to hold such a license. With the A6 Apple started to use a custom ARM design. All newer Apple processors up to the Apple A10 Fusion are based on ARM's IP but with additional SIMD instructions and undisclosed optimizations. The same applies to Qualcomm. Qualcomm already started to design their processors in-house with the Scorpion, the predecessor of their current SoC, the Snapdragon. Thus, both have similarities to an ARM Cortex processor but are effectively custom chips. This means user space applications can be compiled using a generic ARM compiler suit. However, system software developed for an ARM processor might not run on these chips, because both vendors made changes to the core architecture for optimization purposes.

[1] Samsung for example uses in some of its Exynos processors a proprietary interrupt controller.

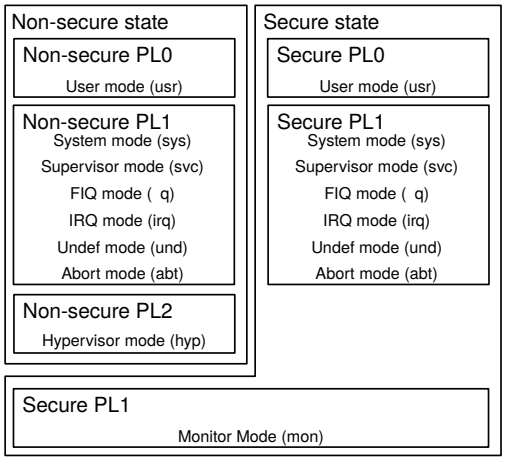


Fig. 2.1: ARMv7 Processor Modes.

Since both designs are proprietary, it is unknown how profound their designs differ from ARM's specification. That being said, in this thesis we only focus on systems with unmodified ARM processors.

In the remainder of the section, we briefly describe important aspects of the ARMv7 processor architecture. All experiments in this thesis were conducted on either one of the following three ARM-based development boards[2]:

- Cubieboard 2, Allwinner A20 SoC (2x Cortex-A7, 1000 Mhz), 1GByte RAM [30]
- Cubietruck, Allwinner A20 SoC (2x Cortex-A7, 1000 Mhz), 2GBytes RAM [31]
- Pandaboard, TI OMAP4 SoC (2x Cortex A9, 1200 Mhz), 1GByte RAM [86]

Therefore, we focus this brief introduction on these processor cores. In particular we describe the execution modes, exception handling, and different processor extensions which provide, e.g., hardware virtualization support.

2.1 Processor Modes

The ARMv7 processor architecture defines seven execution modes (Fig. 2.1). One of these (*usr*) is unprivileged and operates at PL0 (Privilege Level 0), whereas the other six (svc, sys, irq, fiq, und, abt) are privileged and collectively referred to as PL1 (Privilege Level 1). Each change from a lower to a higher PL (Privilege Level) forces the control flow through well-defined entry points [6] called exception vectors. These exception vectors are located in memory. A system control register holds the

[2] In is important to mention that the findings and results in this thesis only apply to systems with an actual ARM core. They may or may not apply to customized ARM-based chips (such as the ones from Qualcomm or Apple).

address that points to this exception vector table. The register VBAR points to the exception vector table for handling transitions from PL0 to PL1[3]. On an exception, the control flow is diverted from PL0 to PL1, where, depending on the reason for the transition (e.g. MMU fault, illegal instruction, system call, interrupt, etc.), execution resumes at the corresponding exception vector (which is an offset from the address where the VBAR register points to).
Each exception vector is 32bit long, thus for most exceptions, e.g., Linux only performs a single branch that jumps to the actual exception handler, which is located somewhere else in memory. Classically there were only two valid locations for the base address of the exception vectors (referred to as the *low vectors* at address 0x0 and the *high vectors* at address 0xffff0000). Since ARMv7 the location can be configured unrestrained. For legacy reasons, e.g., Linux still uses the *high vectors* address location at 0xffff0000. Also, Linux purposely leaves the address 0x0 unmapped to catch null pointer exceptions. Configuring the hardware to use the *low vectors* would prevent the OS from doing this.

2.2 Memory Layout

ARMv7 is a 32bit processor architecture. Thus each processor's physical address space is 4GBytes. Unlike x86 which features I/O ports to communicate with hardware peripherals, an ARM-based SoCs features primarily memory mapped hardware resources. These resources are however not assigned to fixed locations in the address space. Depending on the SoC, hardware resources are placed in different locations inside this 4GBytes physical address space.
When looking at the Cubietruck, which features an Allwinner A20 SoC, e.g., the main memory starts at 0x40000000, and the UART is mapped at 0x01c80000. These addresses are however arbitrarily chosen by the SoC vendor. So, it is important to note, that to develop system software for a particular SoC it is necessary to know these physical addresses in advance. There is nothing like the x86 PCI bus enumeration mechanism to determine which devices are connected to the SoC at which address. Thus, reference manuals and sample code are mandatory when porting system software to a new SoC.
For the ARM architecture, the Linux kernel tried to facilitate the handling of these highly platform and SoC specific characteristics by introducing the DTS (Device Tree Source)/DTB (Device Tree Blob) mechanism. It describes hardware resources assigned to a specific device (e.g., which physical address the device is located at, which interrupts are assigned to the device, etc.). The components are ordered in a tree-like structure in a human readable format. So, these files can also be consulted for the respective information.

[3] The PL2 (described in Section 2.5) has its own copy called HVBAR.

2.3 Coprocessor Interfaces

Apart from the previously described memory mapped hardware resources, there is a number of resources that are addressed using ARM's coprocessor (cp) interface [6]. These are mainly processor components, but also some peripherals, e.g. ARM's architecture timer. ARM provides two 32bit instructions[4] and two 64bit instructions[5] to communicate with these resources. The bits to select the interface in the op-code limits the number of available interfaces to 16.
Currently, ARM only leverages a small number of them anyway, with cp15 as the most prominent one. It provides access to the system control functionality (e.g. access to the `SCTLR` - System Control Register, cache and TLB maintenance, branch predictor configuration, etc.). Other coprocessors (e.g., cp10 and cp11) provide a control and configuration interface for floating-point and SIMD instructions. The cp13 provides an interface to ARMs debugging infrastructure (e.g. configuring breakpoints, halting the system, etc.). All other cp interfaces are reserved for future use. The specific op-code combinations to communicate with each subsystem can be obtained from the ARMv7 reference manual [6].

2.4 TrustZone

On x86 platforms, technologies like TPM (Trusted Platform Module) or Intel's TXT provide means to attest the authenticity of a platform and its OS. With TZ (TrustZone [3, 12]) ARM introduced a similar technology for its processors. But unlike a TPM, which is a fixed-function device, TZ represents a much more flexible approach. ARM's idea with TZ was to leverage the CPU as a freely programmable trusted environment. Fig. 2.1 depicts an ARM processor architecture with the TZ extensions. Orthogonal to the previously described privilege levels, with TZ the architecture is split into two worlds. The new "secure world" effectively duplicates the privilege levels of the classical "non-secure world". Additionally, the monitor mode (mon) was introduced (see Fig. 2.1). It is part of PL1 and was introduced to switch between the non-secure and secure world.
Architecturally, TZ keeps the non-secure world fully backward compatible. The separation of both worlds is mostly implemented in hardware to simplify the design of system software for the secure world. The design of TZ aims at a large degree of autonomy of both worlds, without a perceived need for close interaction. Only the secure monitor call instruction transfers the flow of execution to the mon mode. From there system software can resume the execution in the secure world. Also, the TZ extensions do not allow to configure instruction traps from the non-secure into

[4] `mrc` (**m**ove to **r**egister from **c**oprocessor) / `mcr` (**m**ove to **c**oprocessor from **r**egister)
[5] `mrrc` / `mcrr`

the secure world or any form of nested paging. TZ only provides a coarse-grained memory partitioning[6]. Meaning, parts of the main memory can be marked as secure or non-secure. But many SoC manufacturers use a proprietary TZ controller to hide their IP and encapsulate hardware interfaces for other peripherals through the TZ. Therefore, publicly accessible documentation in regards to TZ controllers is relatively sparse, even though ARM provides a TZ IP core and its specification is open [8].

2.5 Virtualization Extensions

ARM added full virtualization support as an optional feature in ARMv7. Systems with these extensions have an additional execution mode, the HV mode (hyp). This mode is located in the new privilege level PL2, placed below PL0 and PL1, but is only available in the non-secure world (see Fig. 2.1). The design differs from the orthogonal VMX-root/non-root model chosen by Intel and AMD for their hardware VE. PL2 has full access to all system control registers that exist in PL1. But software

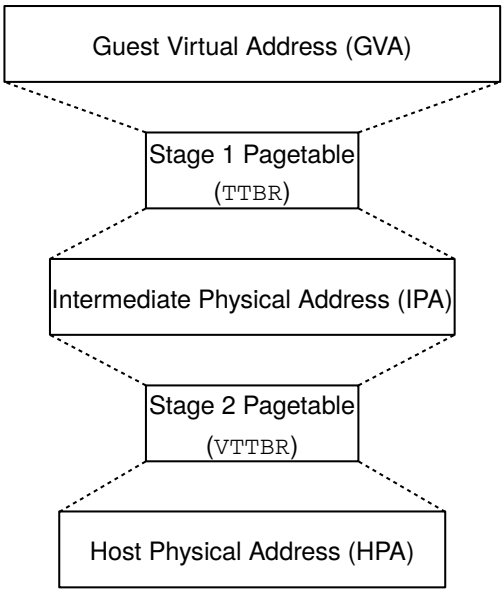


Fig. 2.2: Translation levels on systems with the ARM VE. The stage 1 PT (referenced by the TTBR register) is under VM control and translates from GVAs to IPAs, whereas the stage 2 PT (referenced by the VTTBR register) is under HV control and translates the IPAs to HPAs.

executing in PL2 can configure additional registers to intercept execution in PL0 and PL1, e.g. by inserting additional hardware traps for certain operations.
ARM VE also mandate support for nested paging, a feature that was introduced separately for the x86 architecture. So, with VE two more PTs (Page Tables) are

[6]The granularity of the memory partitioning highly depends on the used TZ controller.

introduced in addition to the existing two used for the secure and the non-secure world. Each PT is referenced by its according register, as shown in Tab. 2.1. Like the secure and PL0/PL1 part of the non-secure world, the address space of PL2 is managed by a dedicated PT, which is referenced by the HTTBR register. Enabling virtualization changes the memory translation regime for PL0 and PL1 by adding a second translation stage. The guest PT, now called stage 1 PT, still translates GVAs (Guest Virtual Addresses) into IPAs (Intermediate Physical Addresses). Instead of putting IPAs directly on the bus, the system subjects them to another translation. The stage 2 PT, referenced by the register VTTBR, is a PT under HV control, translating IPAs into HPAs (Host Physical Addresses). The staged translation scheme allows the HV to assign memory to VMs at page granularity. Fig. 2.2 illustrate the two-stage memory translation regime. Each stage 2 PT entry has its set of permission bits. If

Processor mode	Stage 1	Stage 2
Secure PL0 & PL1	Secure TTBR	
Non-secure PL2	HTTBR	
Non-secure PL0 & PL1 Virtualization active	Non-secure TTBR	VTTBR
Non-secure PL0 & PL1 Virtualization inactive	Non-secure TTBR	

Tab. 2.1: Active PT for different processor modes. The TTBR is a banked register. The secure world and the non-secure world have their dedicated instance. As HTTBR and VTTBR are only used in the non-secure world, they do not need to be banked.

the permissions of stage 1 and stage 2 PT entries contradict, the more restrictive one of the two entries is chosen.

This opens up the opportunity to move security-critical functionality from the complex guest OS into the HV. Also, some security properties are easier enforced on IPAs than on GVAs. For example, while access rights to a page can be restricted (read-only, not executable, either writable or executable but not both, etc.), this restriction is tied to the virtual address that corresponds to the restricting PT entry. The only way to defend against less restrictive memory aliases (references to the same physical page through a different virtual address) is to prevent them from being created. In contrast, stage 2 permissions apply to IPAs and, if more restrictive, overrule possibly permissive guest-controlled stage 1 translations.

3 Related Work

In this chapter, we discuss related work concerning security issues in today's commodity system software. In particular, we focus on two mechanisms. One is build into several commodity systems. The other is part of the Linux kernel. Both, mechanisms revealed severe and still open security issues when examined by researchers. We also highlight related work that proposes security mechanisms, which are embedded into a HV, or microkernel.

3.1 Security of Commodity Systems

Many modern commodity systems (HVs and OSs) leverage a mechanism where memory pages with the same content are merged into a single physical page to save system resources. The feature was introduced to address a problem which often occurs in systems running VMs. A large number of memory pages hold identical content, but the underlying virtualization solution has no way to let VMs share these pages.
To address this issue several OSs and HVs introduced features called *Content-based Page Sharing* [112] (VMWare ESX), *Difference Engine* [55] (Xen), *Kernel Samepage Merging* [5] (Linux) and *Memory combining* [100] (Windows 8 and Windows Server 2012). With these features enabled, the respective system software periodically scans through the main memory to find pairs of pages holding identical content. When it finds such a pair, they are merged into a single page and are then mapped to both locations. The pages are also marked copy-on-write, so the system will automatically separate them again should one process (or VM) modify the content. However, in 2011 Suzaki et al. [103] identified flaws in the feature and consequently were able to exploit them to disclose information of other VMs, effectively breaking the isolation provided by the system software. Based on these findings, Xiao et al. [116] were able to construct a covert channel with bandwidths of up to 1000 bits/s. Further research by Gruss et al. [52] in 2015 and by Bosman et al. [21] in 2016 showed just how relevant this topic still is. Gruss et al. were able not only to determine which applications are running but also identified user activities (e.g. whether the victim currently has a particular website open), once a victim accessed a malicious website and executed some JavaScript code. Bosman et al. showed how to exploit the memory combining feature on Windows 8 systems to disclose

memory locations. Based on the gathered information they mounted further attacks (e.g. based on row hammer).
These examples illustrate that even a simple feature can prompt adversaries to exploit it and break the system's isolation. Moreover, the feature even though known to be vulnerable for several years is still enabled in almost all of the commodity systems mentioned above. The Linux kernel, for example, has the kernel configuration option `CONFIG_KSM` which is set to true, for both x86 and ARM leaving virtualization solutions that are based on the Linux kernel vulnerable to the above-described attacks. Microsoft uses the memory combining feature in all newer versions of Windows. Though, it can be enabled/disabled with the `MMAgent`. Also, the Xen HV still provides this functionality through an optional command line option called `tmem_dup`.

Another common attack scenario on commodity OSs involves placing malicious code or data in user space and then redirecting a corrupted kernel pointer back to the placed user code or data [65]. To prevent the attack, modern processors provide a mechanism called PXN and PAN, respectively which introduce new page table permission bits that allow the OS to mark specific pages as non-executable and non-accessible while running in kernel mode.
But again, a particular design issue in the Linux kernel still allowed researchers to overcome this page protection. In 2014 Kemerlis et al. [64] proved the vulnerability with a novel attack vector. Instead of redirecting a pointer to code or data located in user space they redirect a pointer to point to user code or data aliased in the Linux kernel's *physmap*. The Linux kernel cannot enable the PAN bit for these pages because it frequently needs write-access to them to interact with the user space. However, some regions are also mapped executable which is unnecessary in any case.
Even though in the latest Linux kernel versions this issue has been addressed by mapping each segment of the *physmap* with the correct permissions, there still remains an open issue. The memory page which contains the exception vectors is part of the Linux kernel binary. During boot up the Linux kernel creates a PT alias for this page to point to its dedicated location and marks this PT entry as executable. However, the original page is still part of the *physmap*. This effectively leads to two aliases of the vectors page, one alias marked as executable and one alias as part of the *physmap* which is writable. An adversary would first have to find a way to write to kernel memory (to manipulate the writable alias), but once he was able to find a vulnerability, the fact that this critical page is mapped with these permissions makes the actual exploit much simpler. Once he was able to manipulate the vectors page in the *physmap*, he can execute his code by triggering an arbitrary exception (e.g. `svc`).
Yet again, an issue in the Linux kernel known for two years is not entirely resolved,

giving an adversary an easy way to take over the system software without relying on more complex attack vectors such as ROP or the like.

The above two examples just represent two rather academic threats commodity OSs face. But, the number of more profound attack vectors is much larger. For all major commodity OSs, the CVE (Common Vulnerabilities and Exposures) database reveals an extensive pool of known vulnerabilities [32, 33, 34].

3.2 Virtualization-based Intrusion Detection and Prevention

The general idea of migrating systems into VMs to provide additional security mechanisms is already more than a decade old. The groundbreaking work *"When virtual is better than real [operating system relocation to virtual machines]"* by Chen et al. [25] from 2001 already suggested putting OSs and applications deployed on real machines into VMs. They argue that by relocating an OS into a VM not only provides a compatibility layer to run software for different OSs on a common platform but also provides the option of hosting additional security components isolated from the primary OS. Their initial proposition regarding security mechanisms comprised of secure logging and intrusion prevention and detection. These early thoughts encouraged a large number of researchers to investigate into new forms of security mechanisms based on virtualization.

Soon after Chen's work, Garfinkel et al. [49] proposed the new concept of VMI (Virtual Machine Introspection) in 2003. The work of Garfinkel et al. represented the first virtualization-based IDS (Intrusion Detection System). For this purpose, they modified the VMware Workstation Type-II HV to allow their IDS to inspect the state of the monitored VM. They also designed two components. The first one was an OS interface library which interprets the hardware state exported by the HV and then provides an OS-level view of the VM. The second component was a policy engine consisting of a common framework for building policies, and policy modules that implement specific intrusion detection policies.
In 2007 Jiang et al. [62] coined the term semantic view reconstruction. The main issue with VMI is to bridge the semantic gap between the HV and the guest OS, because the HV has only access to the raw memory of the guest, without any information regarding guest kernel data structures, etc. Therefore Jiang et al. proposed mechanisms to reconstruct the guest VM state from the raw memory. The concept was then transferred to the Xen HV by Hay et al. [56]. They proposed VIX for the Xen HV, which allows for digital forensic examination of volatile system data in VMs. They provided a list of tools (e.g. `vix-ps`), which can be executed in the Dom0. The

tools perform the same tasks as their Unix counterparts but use the raw memory of a DomU in Xen to reconstruct the required information. In [44] Dolan et al. presented an approach for automatically creating introspection tools for security applications, effectively automating the VMI/Semantic View Reconstruction approach. By analyzing dynamic traces of small programs contained in the target system that compute the desired introspection information, they were able to produce new programs that retrieve the same information from outside the target VM. In 2012 Yan et al. [118] transferred the principle of semantic view reconstruction to the Android OS. The architecture named DroidScope uses the emulator Qemu. They extended it with various tracer capabilities to find Malware during runtime.

In 2007, Seshadri et al. [97] were one of the first to propose a HV-based IPS (Intrusion Prevention System). Their thin HV enforces four properties to ensure that only user-approved code is executed in kernel mode. Their architecture called SecVisor only comprises of around $\sim$1500 SLOC and very closely resembles our envisioned system architecture. A similar architecture was proposed by Riley et al. [90] in 2008. An HV-based memory shadowing scheme dynamically copies authenticated kernel instructions from the standard memory to the shadow memory. Any instruction then executed in the kernel space is fetched from the shadow memory instead of from the standard memory. This approach prevents unauthorized code from being executed, thus protecting against kernel rootkits. Again an architecture resembling the one we envision, though leveraging TZ instead of the VE, was proposed by Ge et al. [50] in 2014. Similar to Seshadri et al., they also enforce four properties to rule out a number of attacks on the Linux kernel. A similar approach was taken by Azab et al. [14] who also utilize TZ to ensure guest kernel integrity. Their benchmark results suggest good results, with overhead numbers in the range of $\sim$0.2% up to $\sim$7%.

Part II

Attacks

4 Hardware Virtualization-assisted Rootkits

Similar to the x86 architecture a wide range of rootkits found their way to the ARM architecture [109, 26, 106, 40]. However, on x86 the adversaries did not stop at ring-0 (x86's equivalent to ARM's PL1) to hide their rootkits. Only a year after Intel and AMD released their respective VE, Rutkowska [94] proposed her famous concept of Bluepilling. The attack directly leverages the VE to move a running OS into a VM on-the-fly. Afterwards, a thin HV-based rootkit is installed to control the now victim OS. This way, the rootkit has full control over the OS and is hidden from scanners.

On ARM, on the other hand, only a limited number of rootkits are leveraging such architectural features to cloak their presence [38, 122]. The CacheKit rootkit [122] uses the ARM cache lockdown feature to solely stay in the L2 cache. A rootkit scanner that now scans the main memory is unable to detect the rootkit. However, the L2 cache controller is highly SoC dependent. Only for the Cortex A8 processor is this lockdown feature architecturally specified. For all newer ARM processors (from Cortex A9 onwards), the SoC vendor can decide on which cache controller to use. Furthermore, the CacheKit relies on changing the `VBAR` register. As the address of this and similar structures (e.g. syscall table or vector table) are well known (or even fixed), a rootkit scanner that checks them would recognize the changes (see Chapter 6). Moreover, transferring the Bluepilling concept from x86 to the ARM architecture is not trivial. Unlike the x86 architecture which uses a concept orthogonal to PLs for its VE, the ARM architecture has an additional PL to run the HV software in (see Chapter 2), which is not accessible from PL1. So, the adversary faces the challenge of getting yet another PL down into PL2. Therefore, it is not surprising that such an attack as of yet has not been proposed for the ARM architecture.

In this chapter, we want to address this open research question. We will evaluate whether a truly stealthy HV-based rootkit like the one from Rutkowska [94] is feasible on the ARM architecture. First, we examine the possibility to install a rootkit into the HV mode. Then we assess its detectability.

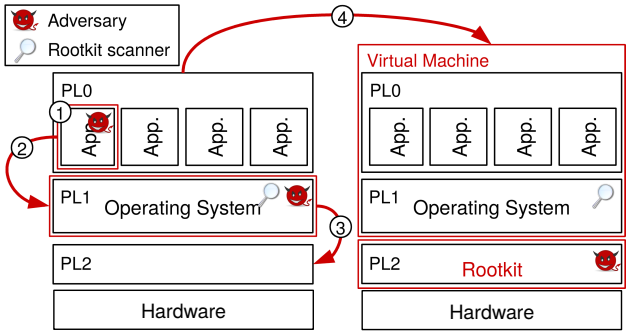


Fig. 4.1: The considered threat model.

4.1 Threat Model

The considered threat model is depicted in Fig. 4.1. An adversary first gains control of a user-level process (Fig. 4.1 ①) or tricks the user into installing a malicious application. Then he manages to exploit a kernel vulnerability (Fig. 4.1 ②). Vulnerabilities in the Linux kernel appear frequently enough [32] to make this a valid assumption. Once having kernel access, the adversary can load his rootkit, but it is then still visible to the OS and exposed to scanners executing directly in PL1 or as a highly privileged process [68, 82, 19]. Therefore, the adversary wants to hide his rootkit by moving it into the even higher privileged PL2 ③. From there, the rootkit can put away the OS into a VM, eliminating the risks of being detected by a scanner in PL1 (Fig. 4.1 ④). During the infection phase, the rootkit is briefly exposed to a scanner running in PL1; however, as we show later in this chapter (see Section 4.5), the time frame is small.

4.2 Entering PL2

The key observation from Section 4.1 is that the adversary must be able to perform the transition from PL1 into PL2 (Fig. 4.1 ③). In the following section, we present several ways to perform this transition and plant malicious code in PL2. It is sufficient to overwrite the exception vector table address of PL2 so that it points to our code. Afterwards, we can trigger an exception from PL1 that traps into PL2 which will execute the planted code. Each of the described attack vectors focuses on overwriting the `HVBAR` register (see Section 2.5). This enables us to gain control on the subsequent PL2 exception.
We want to note that there is no inherent flaw in the ARM architecture. Instead, in many systems, the aspect of locking PL2 is just blithely neglected.

Linux Hypervisor Stub Current versions of the Linux kernel check which mode they were booted into. If they find themself in PL2, they install a stub exception vector table before dropping down to PL1. The purpose of this stub is to allow a Type-II HV implementation (e.g. KVM) to install its own vector table later. It provides support for querying and writing the `HVBAR` register. KVM uses this facility to install its own HV code. All subsequent calls after this installation procedure are then handled by KVM's vector table. Thus KVM has acquired control over PL2 and can use these to control and switch between VMs.
The installation of the stub vector table depends only on the bootup PL. Linux does not provide a way to turn it off. If no KVM module is available or the adversary can mount his attack before KVM is loaded, this provides control over PL2.

KVM Hypercall Function The KVM HV on ARM uses a concept called "split-mode" virtualization [35, 36], i.e., parts of the HV code run in PL1. Only code that explicitly needs access to functionality that is only present in PL2 run in that mode. The component running in PL1 is called "high-visor" and the part running in PL2 is called "low-visor". The "host" Linux is still running in PL1. When KVM is loaded, it installs its own exception vector table, using the HV stub described in the previous section. This prevents an adversary from planting his own code. However, in order to facilitate the communication between low-visor and high-visor, the high-visor component provides a functionality to execute code in the low-visor. The function `kvm_call_hyp` takes a function pointer and will execute the code in PL2. There is no well defined API between low- and high-visor, but upon calling this function in PL1, an aversary can execute arbitrary code in PL2. Yet again, this mechanism can be used to replace the exception vector table.

Migrate Linux Some systems run their rich OS (e.g. Android) completely in the secure world. This facilitates the system deployment because the bootloader does not have to configure the secure world and then switch to the non-secure world. When the system does not need the secure world, this seems like a valid scenario. In the secure world all registers are named exactly the same as their non-secure counterparts (see Section 2.1). Therefore, an OS can either run in the non-secure or the secure world without any changes. But the threat that arises in a scenario where the OS runs in the secure world is the following: on ARMv7, the secure PL1 mode has full control over the mode registers of PL2. Thus, an adversary who manages to gain control over the secure PL1 can modify the PL2 registers. However, for the adversary to gain full control over the OS this is not enough, because the OS still runs in secure PL1 and PL2 only has control over the non-secure PL1. The adversary has to migrate the OS to the non-secure world. Migrating the OS involves duplicating system control register values from the secure to their non-secure counterpart. Also interrupts have to be rerouted to arrive in the non-secure world. After duplicating

the system state and installing malicious code into PL2, the adversary can resume the execution now in the then non-secure PL1.

Vulnerable Secure-world OS Although, the secure world OSs have a reduced attack surface, because of their small TCB and narrow API and might even be audited, researchers still discovered a number of vulnerabilities [92, 99, 13]. So, even if PL2 is properly sealed and none of the above attack vectors is applicable, an adversary can still try to exploit a vulnerability in the secure world OS. Of course the effort is much higher when attacking such a target, but previous attempts have shown that even code execution [92, 99] is possible. An adversary capable of exploiting the secure world OS to gain control over the secure PL1 can configure PL2 and install malicious code.

Texas Instruments Secure API Some TI SoCs (e.g. TI DRA74x) are deployed with a secure world OS in place. The rich OS is able to request services from this secure world OS. Among general functionality, such as cache maintenance, is also an API to install a HV. An adversary in the kernel can abuse this API to install malicious code into PL2. The API works through the dedicated secure monitor call instruction [101]. Upon calling this instruction with a specific ID the execution at a specified location is resumed in PL2. The adversary is then able to install more code into PL2.

Uninitialized PL2 The u-boot [39] is a common bootloader used on a wide range of embedded devices. When compiling the u-boot to run on an ARMv7 SoC it enables the hypervisor call instruction by default[1]. When u-boot now boots the next bootstage (e.g. next bootloader or OS) it drops down into PL2. This allows Linux to install the HV stub as described before. However, this HV stub was introduced in Linux kernel version 3.6-rc6. But many deployed Linux installations run older kernel versions. So, PL2 stays uninitialized, but nevertheless the hypervisor call instruction can be executed and can transfer the execution to PL2. To exploit this, an adversary would need to somehow determine the value of the `HVBAR` register[2]. But the register can only be read from secure PL1 or PL2. To overcome this limitation, the adversary could guess the value of the register. Based on our observations, the value in the register is unpredictable. Still, on a system with 2 GB of RAM, there is a 50% chance of the exception vector table address pointing to main memory if the bit pattern is really uniformly random. If the adversary is able to occupy large parts of the RAM, he can fill it up with a valid PL2 exception vector table and then execute the hypervisor call instruction. Depending on the amount of memory he is able to occupy, there is a good chance that he might hit a valid instruction.

[1] The configuration option in u-boot which is set for all ARMv7 CPUs is called `CONFIG_ARMV7_VIRT`.
[2] The reset value of the `HVBAR` register is undefined [6].

4.3 Hypervisor-based Rootkit Requirements

In order to design a rootkit for PL2 we first identified three requirements such a rootkit would have to fulfill in order to achieve its goals. In particular we identified the following aspects that define the effectiveness of a HV-based rootkit:

- Resilience – The rootkit needs to be resilient and cannot easily be disabled or even deleted by a defender.
- Stealthiness – The rootkit must be stealthy and cannot easily be detected by a scanner residing in a lower privileged execution mode (e.g. PL1 or even PL0).
- Availability – The rootkit must be able to gain control to perform its malicious behaviour and cannot easily be defeated by a DoS attack.

Each point is addressed in the following section.

4.3.1 Resilience

Even though the rootkit executes in PL2 the code pages of the rootkit are memory pages managed by the victim OS. To prevent the victim OS from modifying or removing these pages the rootkit must leverage the staged paging introduced with the ARM VE (see Section 2.5). The stage 2 PT then contains the entire physical address space, except for the pages occupied by the rootkit. However, as the victim OS is unaware that these pages have been repurposed, it might still try to use them. The rootkit must therefore handle these accesses appropriately. To that end, it has various options with different advantages and drawbacks:

- The rootkit could back virtual pages with identical contents with only one physical page, freeing the duplicates for itself. This is similar to the well-established *Kernel Samepage Merging* [5]. Accesses to these pages do not trap and thus perform at native speed; however, the unexpected side-effects of the duplicity of the pages could lead to confusion or a crash of the victim OS. However, direct detection of this behaviour by a scanner is time-consuming, as all pages would have to be scanned in parallel for unexpected write effects.
- The rootkit could leave its own pages unmapped in the stage 2 PT. When the victim OS then tries to access them, it would lead to a stage 2 data abort, which transfers control to the rootkit. The rootkit could now return fake data to the victim OS on a read operation, and ignore write operations to these pages. Accesses to these pages would however be vastly reduced in performance, and a write test would reveal the fake. However, timing effects can be hidden (see Section 4.3.2) and this method can be implemented with minimum complexity.

- Depending on the system, the platform might contain special-purpose RAM besides the main DRAM chips. These are minuscule in size and usually contain small stub routines, e.g., for power management. Depending on the size of the rootkit, it could execute entirely from such an auxiliary RAM. Our investigation on the Cubieboard showed that its SRAM is used by Android to implement different power saving modes. However, the Android standby code does not even occupy a single page of memory, which leaves more than enough room in the ∼64KBytes of SRAM for a rootkit to hide.
- The Linux kernel supports in-kernel memory compression [60]. When this is enabled unused pages are compressed in memory and kept there until the data is needed again. The rootkit could implement a similar feature to compress memory pages and free space for its own pages. Access to these pages would again be reduced in performance, because the rootkit would have to uncompress these pages on demand.

Depending on the purpose and the stealthiness requirements imposed on the rootkit, it can employ one of the above memory handling strategies. Each provides a different trade-off between stealthiness and implementation complexity.

4.3.2 Evading Detection

A sufficiently sophisticated rootkit scanner running in PL1 could detect a rootkit in PL2 in a number of ways. In this section, we discuss the approaches we could employ to obfuscate the rootkit and hide it from a scanners.

Performance Counters The ARM performance counters [6, 7] can be programmed to count instructions executed in a specific processor mode (e.g. hyp mode). They can also be used to count the number of exceptions taken. Both would reveal the presence of code running in PL2. However, the ARM architecture allows the PL2 to trap all coprocessor instructions[3], among them the performance counters. To hide its presence, the rootkit would have to trap and emulate the sensitive performance monitor registers and provide unsuspicious response values. Then the victim OS would still be able to use the performance monitor infrastructure, but the presence of the rootkit would not be revealed.

External Peripherals Mobile devices have a lot of different connected peripherals (e.g. GPS, network card, graphics card, etc.). If the rootkit would, e.g., want to use the network card to exfiltrate sensitive data it would have to make sure that the victim OS can still access the peripheral. If a peripheral is used by both the

[3] The `HDCR.TPM` bit enables trapping of all access to the performance monitoring subsystem into PL2.

rootkit and the victim OS, it might leak state information that could be used to detect the rootkit. In order to avoid this interference, the device either would have to be emulated entirely or the state of the device would need to be reset every time the execution is resumed in the victim OS.

DMA Peripherals Some peripherals have the ability to access memory directly (DMA). A suspecting victim OS could reprogram hardware peripherals to directly write to any physical address, effectively bypassing the stage 2 translation. Such a mechanism threatens the rootkit. On hardware platforms that contain an ARM System Memory Management Unit (SMMU [79]), the rootkit could prevent DMA access to its own pages. It would do so by preventing the victim OS to manage the SMMU, emulating SMMU accesses and then programming the SMMU to restrict DMA access to those pages still available to the victim OS.
On hardware platforms without an SMMU the rootkit would have to emulate every DMA-capable device – third-party DMA controllers as well as first-party DMA devices, e.g. SD/MMC controllers – to prevent its memory from being disclosed or overwritten.

System Emulation Many system control interfaces on ARM platforms are memory mapped (see Section 2.2). For example, the interrupt controller holds the current interrupt configuration state. The victim OS could look at the current configuration and compare it to its expected interrupt state. The rootkit could have, e.g., enabled the dedicated PL2 timer, which it might employ for its periodic execution. The victim OS could discover that. In order to hide these activities, the rootkit would need to emulate the interrupt controller interface as well.

Time Warping As described before, to hide its presence the rootkit could emulate accesses to certain system control interfaces and peripherals. However, a scanner in PL1 would then be able to measure the increased access latencies due to emulation. To prevent this, the rootkit would have to present a virtualized timer to the victim OS. Newer versions of the Linux kernel already use the ARM PL1 virtual timer interface. This allows the rootkit to transparently warp the time for the victim OS.
In case the victim OS uses the PL1 mode physical timer, the rootkit would need to trap all accesses to these timer registers and emulate the “time warp” by reporting lower values. If auxiliary timers (like additional ARM SP804 [9] peripherals) exist on the system, the rootkit would need to emulate the access to those as well. Since the victim OS has no access to an independent clock source on the system, it would not be able to reliably determine how much time has passed since its last measurement. The only chance for a scanner to detect the time drift would be to rely on an external time source (in Section 4.5 we discusses its feasibility).

Cache/TLB Load ARM allows SoC designers to use several levels of caches. But common for current SoCs are just two cache levels, a dedicated L1 cache for each core and a shared L2 cache. To uncover the presence of a rootkit leveraging cache artifacts, a scanner would need to perform several steps. First, the scanner would need to fill up the entire cache with data. Then the scanner would need to wait for a period of time, hoping that the rootkit executes. Afterwards the scanner would need to measure the access times to the data it previously loaded into the cache. If it would measure differences, due to entries being served from the main memory instead of from the cache, the scanner would know that entries have been evicted due to other code being executed. To ensure that no other core caused the eviction of cache lines, the scanner would not only need to halt all other cores but also disable all interrupts. Further, the scanner would need to ensure that no code is executed in other modes (e.g. secure PL1), because this code could also cause data to be evicted from the cache. Depending on the behaviour of the rootkit the system would possibly have to stall for a long time. During normal system execution the scanner would not be able to distinguish whether the cache was filled by an ordinary application or the rootkit in PL2.
The same principle applies to the TLB. When the rootkit executes, it naturally fills the TLB and evicts entries to make space for its own mappings. Moreover, as described in Section 4.3.1 the rootkit might leverage a stage 2 PT to prevent the victim OS from accessing the memory pages of the rootkit. These stage 2 PT translations are cached in a dedicated part of the TLB, the IPA cache. The IPA cache is transparent and fetches translation just like the normal TLB (for stage 1 PT translations), but only for stage 2 PT translations. Thus, under the assumption that the rootkit leverages a stage 2 PT, a scanner would be able to measure artifacts originating from IPA cache hits or misses. However, certain aspects of the IPA cache design could still prevent a scanner from detecting the rootkit. To detect a rootkit that leverages the stage 2 PT, the guest would first need to fill the IPA cache entirely. However, only the page granularity the rootkit chooses for its stage 2 mappings (4KBytes, 2MBytes or 1GByte) decides whether the IPA cache can be overflown at all. Whether or not a rootkit maps the memory using large pages (e.g. 1 GB) depends on the rootkit's strategy of avoiding detection (see Section 4.3.1).

4.3.3 Availability

Finally, as every other rootkit, a rootkit in PL2 must periodically gain control to perform its malicious operation. We came up with two modes of operation for the rootkits which we termed *proactive* and *reactive* execution. Whether a rootkit operates in *reactive* or *proactive* mode has again implications on stealthiness, runtime and implementation complexity:

Proactive execution In the *proactive* execution mode, the rootkit would require a time source to periodically gain control. A periodic timer interrupt that is routed to PL2 can be configured, so that the rootkit is able to perform its malicious operation. The interrupt controller however does not provide a mechanism to selectively route interrupts to different privilege levels. Therefore, in the *proactive* model, the rootkit would need to intercept all interrupts. The rootkit then would need to filter out its PL2 timer events and deliver all other interrupts to the victim OS. This approach is more complex to implement and increases interrupt latency, but it is perfectly suited for data exfiltration attacks where keystrokes or other user actions are monitored during phases of platform activity and later transmitted to an external command-and-control entity when the platform is otherwise idle.

Reactive execution The *reactive* execution is less invasive, because the rootkit would only react to certain stimuli from inside the victim OS. However, most traps that can be configured to target PL2 can only originate in PL1 (and not PL0), e.g. the hypervisor call instruction. Execution of such an instruction in PL0 is considered undefined and would simply be reported to PL1. One of the few exceptions is trapping the deprecated Jazelle[4] instructions. These instructions can directly trap from PL0 to PL2. The ARMv7 specification mandates that any system implementing the VE provides only the trivial (i.e. empty) Jazelle implementation. This implementation only includes some Jazelle control registers and the `bxj` instruction. It also mandates that `bxj` must behave exactly like a `bx` instruction. However, an ARMv7 processor still provides a means to trap attempts to access Jazelle functionality to PL2. Thus, in the *reactive* execution mode the rootkit would enable trapping of the `bxj` instruction into PL2. Now PL0 application would be able to trigger a PL2 exception by executing a `bxj` instruction.

The *reactive* approach is much easier to implement than the *proactive* model, and it has almost zero overhead during regular system activity. However, it is more suited for externally triggered attacks. For example, an unsuspicious application with network connectivity could allow an adversary to invade the platform, quickly elevate his privileges by activating the rootkit, steal sensitive information, and deprivilege itself again all by signalling the rootkit with the `bxj` instruction.

4.4 Design Criteria & Implementation

Based on the previously defined requirements on *resilience*, *detectability* and *availability* we designed a HV-based rootkit. This proof-of-concept implementation

[4] Jazelle is a special processor instruction set for native execution of Java bytecode found in earlier ARM cores.

consists of the code that runs in PL2, a Linux kernel module, and two user space applications. All components are described in the following section.

Of course a real attack would first contain the transition from PL0 to PL1 (see Section 4.1), which would rely on a real vulnerability in the Linux kernel. For simplicity we implemented a kernel module to load our rootkit code directly into PL1. The kernel module provides a device node where we supply our rootkit binary, along with a number to signal the kernel module which attack vector to use. The kernel module then exploits the specified attack vector to deploy the rootkit into PL2.

Once the rootkit is deployed its execution is split into two parts. The *initialization phase* starts immediately when it is loaded. Depending on the attack vector the *initialization phase* starts in secure PL1 (Attack vector 3) or directly in PL2 (Attack vectors 1 and 2). After the initialization phase the rootkit enters *runtime phase*, where it provides its malicious service.

4.4.1 Initialization Phase

After the rootkit is loaded into secure PL1 or PL2, respectively, it has to perform a number of operations:

1. Migrate to non-secure mode (Attack vector 3 only).
2. Setup a stage 2 PT.
3. Activate traps of emulated registers.
4. For *proactive* execution: Configure the interrupt controller and the PL2 mode timer.

In the first (optional) step the rootkit checks whether the processor's current security state is secure. It then migrates the current setup to the non-secure mode. To do so, all register are copied from the secure to their non-secure counterparts. Additionally, the interrupt controller is configured in a way that all interrupts are routed to the non-secure world. To allow the non-secure world to access coprocessor registers, the `NSACR` register is configured to allow non-secure access to all coprocessors. Once the migration is finished, the initialization code goes over to step 2, the setup of a stage 2 PT.

For our rootkit we decided to go for the first approach discussed in Section 4.3.1. Our rootkit creates a stage 2 PT which contains translations for the entire physical address space except the memory pages that contain the rootkit itself. Any access from the victim OS to a page occupied by the rootkit will result in a trap into PL2,

which we then emulate. Once the stage 2 PT is setup and activated, step 3 is performed.

As already described in Section 4.3.2, certain performance monitoring registers can be configured to reveal the presence of the rootkit. Therefore we trap all accesses to these registers and emulate their behavior. This way, we are able to filter out critical events.

This last step is optional, depending on whether the rootkit runs in *reactive* or *proactive* mode. However it also has an influence on the layout of the stage 2 PT. In *reactive* mode the rootkit does not need access to the interrupt controller at all, so it can just forward the interfaces to the victim OS. However, when running in *proactive* mode the rootkit has to adjust the memory layout. The interrupt controller is memory mapped. Thus the rootkit must make sure to not provide the actual interrupt controller interface to the victim OS. Instead, the rootkit maps the virtual interrupt controller interface to the address where usually the normal interrupt controller interface in the victim OS's address space resides. Then, the rootkit copies the complete state of the interrupt controller to the virtual interface and then enables the virtual interface. Finally, the rootkit enables the PL2 timer to gain periodic control.

4.4.2 Runtime Phase

As discussed in Section 4.3.3 we implemented both modes of operation *reactive* and *proactive*. The implications on the overall system performance based on the execution mode are provided in Section 4.5.

Independent from the fact whether the rootkit runs in *proactive* or *reactive* mode, a number of operations need to be done. First, the cycles the CPU spends in PL2 must not be visible to the victim OS. Recent versions of the Linux kernel already use the virtual timer infrastructure, which makes it easy to warp the time for the victim OS. The rootkit warps the guest timer in the following manner: upon each entry into PL2 mode the current time value is saved. Upon exiting PL2, the rootkit again reads the current time value. The gap between these values is then stored in the appropriate offset register. The ARM virtualized timer infrastructure automatically subtracts the value of this offset register whenever the victim OS reads its "virtual" time. Thus, the time spent in PL2 is no longer detectable from PL1.

In addition to the time warping, which is necessary in both modes of operation, in *proactive* mode, the rootkit also has to handle interrupts. In order to use a dedicated timer for PL2, all interrupts must be trapped into PL2. Upon each interrupt the rootkit checks whether the interrupt originated from the PL2 mode timer or not. In the latter

case, the interrupt is simply forwarded to the victim OS; otherwise the rootkit handles the interrupt itself and performs its malicious operation. Afterwards, execution is resumed in the victim OS.

4.5 Evaluation

The effectiveness of any rootkit heavily depends on its stealthiness. As described in Section 4.3, some transitions from PL1 into PL2 are inevitable. Thus, in this section we evaluate how long certain operations take and discuss the effectiveness of scanners trying to detect the presence of the rootkit. All tests were conducted on a Cubieboard 2 [30].

4.5.1 Startup

While the rootkit is in its initialization phase, it is exposed to rootkit scanners as the memory pages containing the rootkit are present in the victim OS's memory view. However, our measurements show that the startup time for our rootkit is only $\sim$0.18ms. A scanner that searches the memory for suspicious content would only be able to detect the presence of the rootkit within this time frame. As the scanner can not make any assumptions about where in the memory the rootkit is located, this time frame is sufficiently small to remain stealthy in the presence of such rootkit scanners.

4.5.2 Benchmarks

In the runtime phase a rootkit scanner could try to uncover the rootkit through the induced performance overhead (e.g., when the rootkit runs *proactive* all interrupts cause the CPU to trap into PL2). Also the 2 stage address translations induces overhead that a scanner could try to measure. To estimate the effectiveness of such a scanner we performed a number of standard system benchmarks. With two benchmarking suites (lmbench [78] and hackbench [123]) we measured the rootkit's impact on these low level operations. Table 4.1 shows the results. Column 1 describes the performed benchmark, the other columns show the results in the respective setups. We performed each benchmark 50 times and calculated the mean values and their respective standard deviation. The mean values show a slight, but noticeable performance overhead in the rootkit setups. However, the high standard deviation values render the mean value difference almost undetectable.

Benchmark	Linux		rootkit *(proactive)*		rootkit *(reactive)*	
	mean	std. dev.	mean	std. dev.	mean	std. dev.
`lat ctx 2`	58.1050	4.8200	59.2100	5.6957	58.8400	4.6556
`lat ctx 4`	64.3100	4.3080	65.8950	5.3935	65.8300	4.3352
`lat ctx 6`	64.1968	4.7098	65.3250	5.7011	65.6696	4.4189
`lat ctx 8`	66.0458	4.4091	68.2240	5.1715	67.6644	4.3407
`lat ctx 10`	68.1260	4.8018	69.6390	5.8112	69.2080	5.1669
`lat syscall`	0.2785	0.0018	0.2787	0.0007	0.2785	0.0014
`lat read`	0.6623	0.0015	0.6628	0.0015	0.6625	0.0015
`lat write`	0.4779	0.0009	0.4788	0.0010	0.4781	0.0009
`lat pipe`	12.5509	0.6583	12.6093	0.7291	12.8524	0.8827
`lat select`	15.7479	0.0061	15.7526	0.0074	15.7502	0.0076
`hackbench`	3.1301	0.0123	3.1352	0.0129	3.1301	0.0153

Tab. 4.1: lmbench and hackbench benchmarking results (lmbench benchmark results are in microseconds and hackbench results are in seconds).

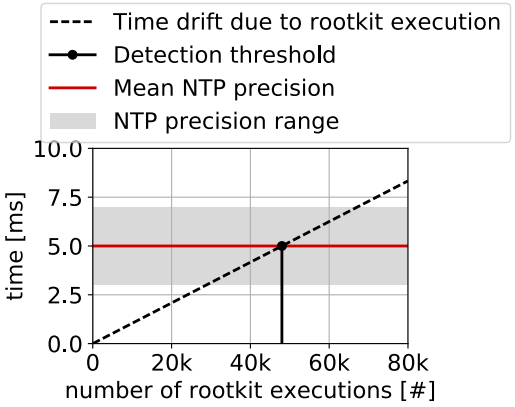


Fig. 4.2: Detectability of our rootkit in *reactive* execution based on time drift.

4.5.3 Clock Drift

Another approach is to measure the clock drift that is induced by the rootkit. As described in Section 4.3.2 the rootkit can hide the clock cycles that the CPU spends in PL2. Still, in combination with an external time source a scanner could try to detect the time drift between the local clock and the external clock. Since the scanner can not know when the rootkit actually executes, it would rely on blindly enforcing traps into PL2 to reveal the clock drift. This could be done by e.g. multiple executions of a `bxj` instruction in the *reactive* setup or by utilizing a peripheral to trigger a large number of interrupts in the *proactive* setup. Fig. 4.2 depicts the drift of the local clock compared to an external clock, e.g. NTP. Assuming an NTP accuracy of $\sim$5ms over an internet connection the clock drift introduced by the rootkit becomes visible after 60.000 traps into PL2, which could be either an execution of `bxj` or an interrupt handled by the rootkit.

In both cases, a huge number of events is necessary in order to build a scanner that could reliably discern between a native and a rootkit-infected system. Although not implemented by us, we argue that the rootkit could be retrofitted with an "alarm mechanism" that detects unusually large numbers of PL2 entries and activates appropriate countermeasures to evade detection (e.g. switching from *proactive* to *reactive* execution).

5 Breaking Isolation through Covert Channels

Covert channels are a well-known threat not only in high-security systems but also in cloud scenarios [85, 117, 116] or on mobile devices [74, 24]. With a covert channel, adversaries can covertly exchange data between two entities on a platform or exfiltrate data to an external agent. The issue came first to public attention when Lampson described the problem in 1973 [71]. In 1987, Millen [80] came up with a theoretical approach to estimate the capacity of covert channels and the US Department of Defense acknowledged the threat in 1993 with a classification scheme for covert channels [1].

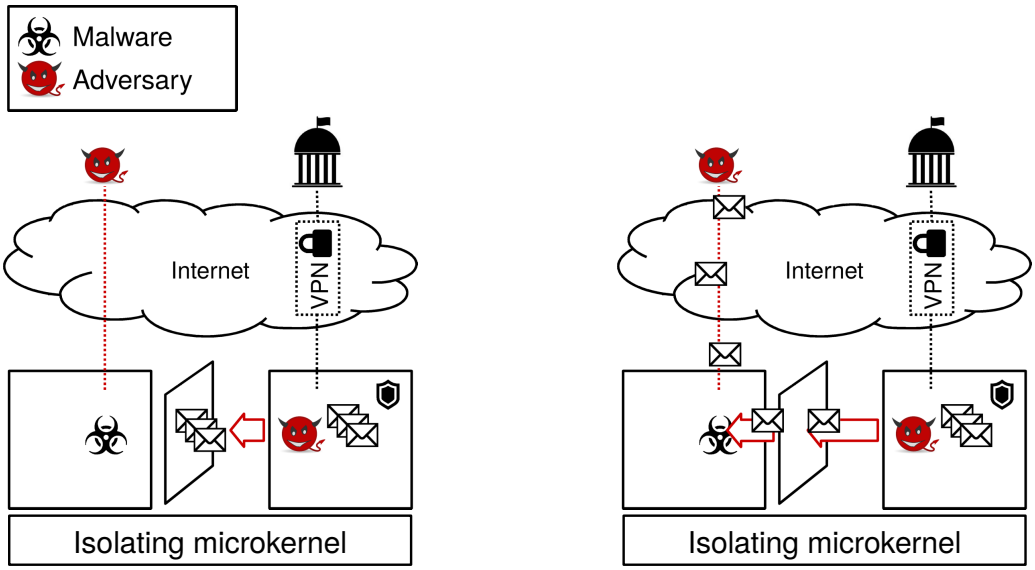
On traditional system architectures (e.g. Linux), something as simple as a file or a process ID [74] can be used to form a covert channel between two entities in the system. The topic witnessed a renaissance when the research direction shifted towards virtualization because the issue is especially delicate in the cloud scenario where users run software in potentially untrusted environments. There, shared hardware resources (e.g. cache, RAM, TLB, etc.) naturally leak information to other entities with access to the same medium [53, 91] and provide excellent means to form a covert channel.
But also deficiencies in the underlying virtualization solution can lead to covert channels [103, 116]. For example, the *memory deduplication* feature in modern virtualization solutions is exploitable [103], such that it allows to create very high bandwidth covert channels. Thus, alternative system architectures such as microkernels are considered for security critical environments. Their small trusted computing base and isolation properties promise more security than traditional architectures and fewer options to form covert channels.

In order to validate that very promise, in this chapters we investigate Fiasco.OC a microkernel of the L4-family. Contrary to the promised isolation properties we uncover weaknesses in Fiasco.OC's kernel memory subsystem. Our findings allow to undermine all efforts to isolate components in security critical systems built on top of Fiasco.OC. Subsequently, we develop real-world covert channels based on those weaknesses found in Fiasco.OC's kernel memory management. Following Millen [80] we measure the capacity of the respective channels to gain a better understanding of their applicability and severity.

5.1 Attack Model

First, we formulate the assumptions regarding the underlying system architecture and the assumed capabilities of the adversary. We envision a system as depicted in Fig. 5.1 with (at least) two compartments where the communication between them is understood to strictly obey a security policy. The policy may state, e.g., that no communication between any two compartments shall be possible. The corporate policy may require that assets are only sent through a secure connection and can only be processed in a compartment without direct Internet connectivity. Under these provisions, the confidentiality of assets should be preserved as long as the isolation between compartments is upheld.

As for the adversary, we consider highly determined adversaries who managed to place malware into all compartments. To achieve this, the adversaries may either directly attack Internet-facing compartments or draw on insider support to sneak in the malware. The adversary has the goal to leak the assets from the isolated compartment to a compartment with access to the Internet. From there, he can send the assets to a place of his choice.



(a) Effective isolation.

(b) Ineffective isolation.

Fig. 5.1: An effectively isolating microkernel can prevent data from being passed between compartments, even if both of them have been compromised by an adversary (Fig. 5.1a). If the microkernel is ineffective in enforcing this isolation, data may be first passed between compartments and then leaked out to a third party in violation of a security policy prohibiting this (Fig. 5.1b).

Regarding the system, we assume that the microkernel has no low-level implementation bugs and the system configuration is sound. The system configuration also does not permit direct communication channels between separate compartments.

5.2 Fiasco.OC Memory Management

Fiasco.OC [46] is a microkernel developed at the TU Dresden (Germany). It is distributed under the GNU General Public License (GPL) v2 and runs on x86, ARM, and MIPS-based platforms. It comprises of 20kSLOC to 35kSLOC, depending on the system configuration. A security model based on capabilities support the construction of secure systems. As many other microkernels, Fiasco.OC is accompanied by a user-level framework called L4Re [70], which provides both libraries and system components aiding in the construction of highly compartmentalized systems. As many other L4-like kernels, Fiasco.OC separates its user and kernel memory management. The entire user-level memory management is handled outside the kernel. Special root components called Sigma0 and Moe provide initial resources and exception handling to bootstrap the system. User-level servers can then implement memory management policies depending on the needs of the applications at hand. This way the complexity of the kernel is reduced. To that end, Fiasco.OC provides three mechanisms:

1. Page faults are exported to user-level, usually by having the kernel synthesize a message on behalf of the faulting thread.
2. A mechanism whereby the right to access a page can be delegated (L4 terminology: *map*) between tasks.
3. A mechanism to revert that sharing (*unmap*)[1].

The situation is, however different for kernel memory, for which Fiasco.OC does not provide a direct management mechanism. When user processes want to create a new kernel object (e.g. IPC gate, thread, etc.) the kernel turns to its internal allocators to request memory to create the according object. To prevent users from monopolizing this resource (e.g. by creating a large number of these objects), Fiasco.OC implements a quota mechanism to divide the kernel memory among tasks in the system. Every task is associated with a quota object, which represents the amount of kernel memory that is available for all activities in that task. Whenever a user activity, e.g. a system call prompts the kernel to create a kernel object, the kernel first checks whether the current quota covers the requested amount of memory. If this is not the case, the system call fails. For each initial task the amount

[1] Since pages can be recursively mapped, the kernel needs to track this operation, otherwise the unmap might not completely revoke all derived mappings. The kernel data structure used for this purpose is called mapdb, an abbreviation for mapping database.

of available kernel memory is specified in a startup script. The system integrator has to specify quota values whose sum is not larger than the amount of kernel memory available to the system at hand. Every task is free to split its share of kernel memory and supply further tasks with it. Thus, the process can be repeated recursively.

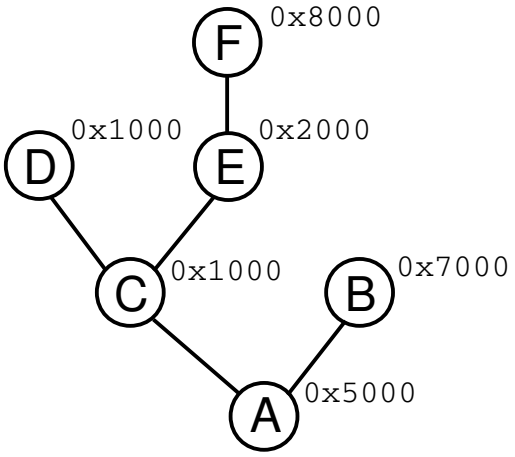
5.2.1 Kernel Allocators

At the lowest level, kernel memory is managed by a buddy allocator. Depending on the amount of main memory in the system, 8% to 16% of the system's memory is reserved for kernel use and fed into the kernel's buddy allocator. Because the size of kernel objects is not a power of two and also differs among objects, allocating them directly from the buddy allocator would cause fragmentation over time. Therefore many objects are not directly allocated from the buddy allocator. Instead, they are managed through slab allocators, which in turn are supplied with memory from the buddy allocator. Every slab allocator, eighteen in total, accommodates only objects of the same type.

5.2.2 Hierarchical Address Spaces

As described before, Fiasco.OC tracks physical pages that are used as user memory in a structure called mapdb. To store this information efficiently, Fiasco.OC uses a compact representation of a tree in a depth-first pre-order encoding. Every mapping can be represented by two machine words holding a pointer to its task, the virtual address of the mapping and the depth in the mapping tree[2]. Fig. 5.2 illustrates the principle. While this representation saves space compared to other implementations using pointer-linked data structures, it brings about an object that potentially grows and shrinks considerably depending on the number of mappings of that page. If the number of mappings exceeds the size identifier of the mapping tree, the kernel allocates a bigger tree, into which it moves the existing mappings along with the new one. Conversely, when a thread revokes mappings, the kernel invalidates tree entries. Shrinking the tree, which involves moving it into a smaller data structure, takes place when the number of active entries is less than a quarter of the tree's capacity.

[2] Since a page address is always aligned to the page size, the lower bits of the mapping address can be used for the depth in the mapping tree.



id	depth	virt. address
A	1	0x5000
C	2	0x1000
D	3	0x1000
E	3	0x2000
F	4	0x8000
B	2	0x7000

Fig. 5.2: Mapping tree layout.

5.3 Unintended Channels

In this section, we present so far undocumented issues with Fiasco.OC's kernel memory management, which can be exploited to open up unintended communication channels. They were not anticipated by the designers and hence cannot be controlled by existing mechanisms. As a result, no security policy can be enforced on them.

5.3.1 Allocator Information Leak

As described in Section 5.2, appropriately set quotas should ensure that each task only consumes the share of kernel memory allocated for it regardless of the activity of other tasks. However, due to fragmentation, it may happen that the allocators cannot find a contiguous memory range sufficient to accommodate the requested object. We will show that the combination of Fiasco.OC's design and implementation gives an adversary the opportunity to fragment the kernel memory on purpose. This allows him to tie down kernel memory far beyond what his quota should allow, effectively rendering the quota mechanism useless.

While the quota accounts for objects created on behalf of an agent, it does not capture the unused space in the slabs. It can be assumed that object creation is random enough that over time all slabs are evenly filled and that configuring a system with only half of its memory made available by quotas properly addresses the issue of fragmentation. Yet an adversary is capable of causing a situation where this empty space accounts for more than 50% by deliberately choosing the order in which objects are created and destroyed. A graphical illustration of the process is provided in Fig. 5.3. To illustrate the point, both the adversary's quota and the amount of used kernel quota are assumed to be zero. To accommodate the first

object, the kernel allocates a new slab which is then used for subsequent allocations of that type (Fig. 5.3 ①). Then the adversary allocates more objects of the same type so that the kernel has to allocate a second slab (Fig. 5.3 ②). It is crucial to understand which objects are getting released when the adversary destroys a kernel object. The Fig. 5.3 ③ shows two possibilities. If the two remaining objects reside in the same slab, the second slab is not needed anymore, and its memory can be reclaimed by the system allocator. In case the objects are allocated in different slabs, both slabs have to be kept. If repeated, the adversary can cause the system to enter a state where it is filled with many slabs with only one object. While allocation requests for objects whose slab caches have nearly-empty slabs can be served easily, no new slabs for other objects can be allocated.

If objects could be shifted between slabs, system resources could easily be reclaimed (defragmentation). The memory of the newly freed slabs could be returned to the underlying system allocator, solving the problem. But, Fiasco.OC does not possess such a defragmentation ability.

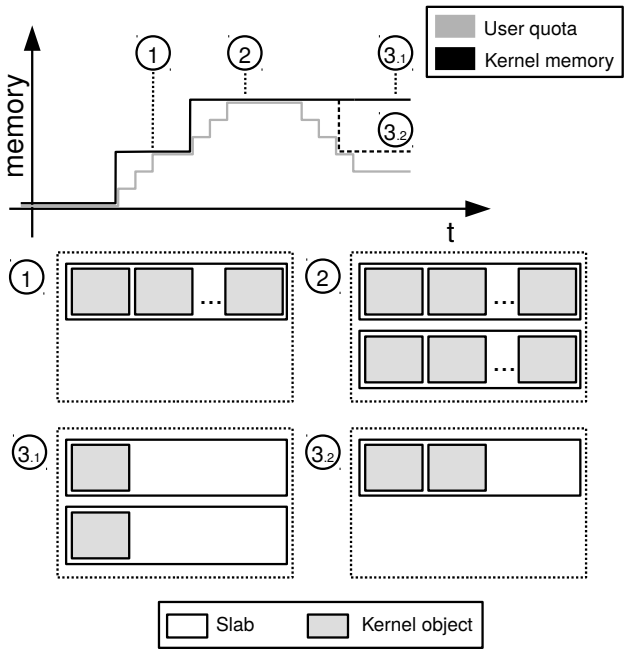


Fig. 5.3: Object placement in slabs. Depending on the order in which objects are created and destroyed, the number of slabs used to accommodate them can vary.

The amount of memory that can be tied down depends on four factors. The number of vulnerable slab caches. The number of objects held in their individual slabs The order in which slabs are attacked, and, for objects with variable size such as mapping trees, on the minimal size required for the object to stay in a certain slab. In our experiments, we were able to tie down six times the amount of the assigned

quota. In a system with two tasks where the available kernel memory is equally divided, a factor of two is enough to starve the system completely.

5.3.2 Mapping Tree Information Leak

In addition to the allocation channels described in the previous section, there is another implementation artifact that can be misused as a communication channel. As outlined in Section 5.2.2, Fiasco.OC tracks user-level pages in tree-like data structures, so-called mapping trees. Available in various sizes, the mapping tree data structure grows and shrinks as the page it tracks is mapped into and unmapped from address spaces. During an unmap, the kernel uses it to find the part of the derivation tree that is below the unmapped page, if any and unmaps it as well. Although the mapping tree structure is dynamically resizable, Fiasco.OC limits the number of mappings that can exist of a physical page to 2047.
At first glance, this might not seem to be an issue because isolated processes are not meant to share pages. However, L4Re, the user-level OS framework running on top of Fiasco.OC, provides different services, among them a runtime loader. This loader, which is not unlike the loader for binaries linked against dynamic libraries on Linux (ld.so), maps some pages into the address space of every process. Experimentally, we found that 18 pages are shared this way among all processes that are started through the regular L4Re startup procedure.

5.4 Channel Construction

In this section, we explain how to construct a high bandwidth covert channel. We also devise more sophisticated channels to work around problems impeding stability and transmission rates, which allow us to establish a reliable and fast covert channel even in difficult conditions. Three of the proposed channels rely on the fact that a malicious process is able to use up more kernel memory than its quota allows. The remaining channel exploits the limit in the mapping tree structure.

5.4.1 Page Table Channel

The Page Table Channel (PTC) requires an initial preparation as described in Section 5.3.1. Following this, the channel can be modulated in two ways: for sending a 1, the sender allocates PT from the kernel allocator. For transferring a 0, it waits for one interval. To facilitate PT allocations, the sender creates a helper task and maps pages into its address space. The sender places these pages in such a way that each page requires the allocation of a new PT. The receiver can detect

the amount of free memory in the kernel allocator by performing the same steps as the sender. The number of PT available to the receiver is inversely proportional to the number of PT held by the sender. This knowledge can be used to distinguish between the transmission of a 1 and a 0. At the end of every interval, the sender has to release the PTs it holds. Unmapping the pages from the helper task is not sufficient because Fiasco.OC does not release PTs during the lifetime of a task. Instead, the helper task has to be destroyed and recreated. The implications of this will be discussed in detail in our evaluation in Section 5.7.

5.4.2 Slab Channel

The Slab Channel (SC) uses contention for object slots in slabs to transfer data. For this purpose, we set up the channel as described in Section 5.3.1 and subsequently perform channel specific preparations. One slab is selected to hold mapping trees for the transmission. This slab is filled with mapping trees until only one empty slot remains, which sender and receiver then use for the actual communication. Fig. 5.4 shows the principle. To transfer a 1, the sender needs to fill this last slot. Assuming a slab of size 4KB for the transmission, it causes a mapping tree residing in a slab of 2KB to grow, until it exceeds its maximum size. The kernel will then move it into a bigger slab – the one intended for transmission. The receiver can determine which bit was sent by performing the same operation as the sender. If this fails, the receiver interprets this as a 1 and as 0 otherwise.

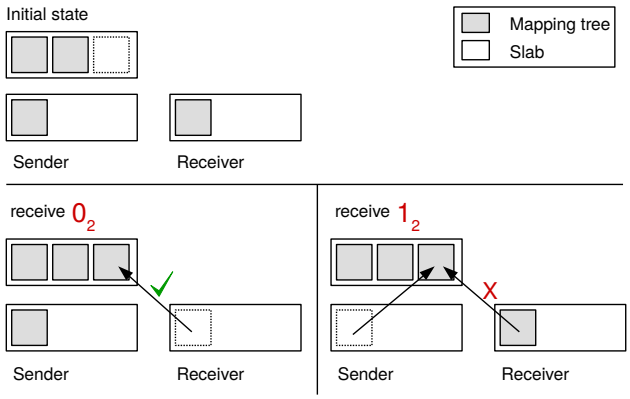


Fig. 5.4: Transmission in the slab channel. The sender fills or leaves empty the last spot in a slab; the receiver reads the value by trying to move an object into that spot and checking the return code indicating the success or failure of the operation.

5.4.3 Mapping Tree Channel

As described in Section 5.3.2, L4Re maps 18 read-only pages into each application's address space. Like every other page, a mapping tree (Section 5.2.2) tracks these pages. The Mapping Tree Channel (MTC) exploits this fact.

When the mapping tree reaches its maximum depth of 2048, any attempt to create new mappings of the page will fail. In the most basic form, the communication partners agree on one shared page, which they then use as a conduit. In the preparatory phase, one of the conspirators fills the mapping tree of the chosen page such that only room for a single entry remains. Creating these mappings is possible because the shared pages are regular and are not subject to mapping restrictions[3]. To transfer a 1, the sender creates a mapping of the chosen shared page, maxing out its mapping count. The receiver, concurrently trying to create a mapping, fails as a result. Conversely, by not mapping the page, the sender transfers a 0, so that the receiver's operation succeeds. Compared to the other two channels, the MPC incurs low overhead. Unlike the SC, an MPC transmission requires at most three operations. In contrast to the PTC, no costly task destruction is necessary.

5.5 Channel Optimizations

Since, Fiasco.OC's 1000Hz timer resolution limits the step rate of clock-synchronized channels, the only path to higher bandwidths is to increase the number of bits transmitted per step. To that end, we devised two methods that allow for increased channel bandwidth at the cost of higher CPU utilization.

One of the conspirators can trivially boost the bandwidth by increasing the number of sub-channels. The underlying assumption here is that several of them are available. Fiasco.OC maintains multiple slabs, each of which can constitute a channel. An adversary can now choose between using a slab for tying down kernel memory or use it as a communication channel. Likewise, the adversary can exploit multiple of the existing shared pages.

A different approach is to transmit multiple bits per channel and step. For that, a channel needs to be capable of holding 2^n states, n being the number of bits[4]. Channel levels can be realized by occupancy levels, assuming the adversary can allocate the underlying channel resources in increments. The number of operations

[3] There are regions in Fiasco.OC tasks, such as the UTCB that, while being accessible, cannot be mapped.

[4] The levels can be spread over multiple channels, so as to be more flexible regarding the levels required per channel.

to bring up and sense these levels grows exponentially with the number of bits. For that reason, multi-channel schemes, where the number of operations increases linearly with the number of bits, might be preferable. However, the number of available sub-channels is limited.

5.6 Transmission Modes

Under clock synchronization, two agents make use of a shared clock to synchronize their execution. The sender and receiver share a notion of points in time where they have to be done with certain actions. It is the responsibility of either party to make sure that its activity (writing to the channel or reading from it) is completed before the next point is reached as there are no additional synchronization mechanisms whereby the other party could detect that its peer's activity was not finished.
Fiasco.OC provides a sleep mechanism with a 1ms wake-up granularity. Moreover, Fiasco.OC exposes the current system time through the KIP (Kernel Interface Page) a page that can be mapped into every task. This global clock is very helpful for scenarios with long synchronization periods that consist of multiple 1ms ticks.
On systems under heavy load, there is no guarantee that conspiring threads are executed frequently enough as they compete with other threads for execution time. Whenever either party misses an interval, bits are lost, and the transmission becomes desynchronized. To alleviate this problem, we can either incorporate error correction into the channel – at the cost of the bit-rate – or we can design our channel to be independent of a common clock source.

Under the self-synchronizing regime, sender and receiver do not observe a shared clock. Instead, they dedicate some of the available data channels to synchronization, effectively turning them into spinlocks. Using this mechanism, we can ensure that sender and receiver can indicate to each other whether the other party is ready to write to or from the channel.
One drawback of self-synchronization is that at least two data channels have to be set aside for lock operations, reducing the number of the channels for data transmission. Especially in setups where data channels are rare, this can be a serious issue. We take a look at the achievable bit rates in Section 5.7.2.

5.7 Evaluation

To evaluate the feasibility of our approach and to measure the achievable bandwidth of the various channel configurations, we ran a number of experiments on a

Pandaboard (for a detailed specification of the board refer to Chapter 2). For all experiments we used Fiasco.OC/L4Re [46, 70] version r54.

In our measurement setups, we use two agents (sender and receiver) implemented as native L4Re applications. Our setup does not permit any direct communication channel between them. Unless stated otherwise, the sender transmits packets of four bytes. The first byte holds a continuously-incrementing counter value followed by three bytes with a CRC checksum. The receiver feeds the received bits into a queue and checks whether the latest 32 bits contain a valid CRC checksum. All transmission rates reported in this section reflect the number of bits transmitted per time interval, including the checksum bits. The "channel states" row in our tables indicate the number of states n a channel needs to support to transmit $\log_2 n$ bits per interval.

5.7.1 Clock-synchronized Transmission

In this section, we evaluate all clock-synchronized channels (Section 5.5). We first assess the basic capacities we observed and then investigate the effects of individual improvements, such as transmission with multiple channels or multiple bits at a time. Finally, we take a look at the CPU utilization of our transmissions and what conclusions we can draw from these numbers.

Our first setup implements the PTC (Section 5.4.1) as our most basic transmission method. During our experiments, we noticed that enabling SMP support results in lower bit rates when compared to UP setups. This counter-intuitive observation can be explained by taking a closer look at the time required to perform the individual operations for sending and receiving data through the PTC. On SMP enabled setups, the destruction of the helper task takes significantly longer. The reason lies within the internals of Fiasco.OC. Destroying a task in an SMP enabled configuration employs an RCU [77] cycle (a synchronization mechanism which incurs a small overhead while also being simple). The downside of this approach is an additional latency for individual operations because it can only terminate after a grace period has elapsed. In Fiasco.OC, this grace period is 3ms long. In an SMP setup, this affects clock-synchronized transmission methods that involve frequent object destructions, such as destroying a task. The PTC, in particular, suffers from this circumstance and achieves on an UP system more than twice the bitrate compared to an SMP system. Tab. 5.1 therefore only contains UP results. With the PTC we are able to transmit data at a constant rate of 500bits/s.

The SC (Section 5.4.2) operates in principle similar to the PTC but does not require a potentially costly object destruction. The transmission rate is solely limited by the

Channel	Channel states (#)	Period (clock ticks)	Throughput (bits/s)
PTC	2	2	500
SC	2	2	500
MTC (2 channels)	2	2	1000
MTC (8 channels)	2	4	2000

Tab. 5.1: Capacity results for the three basic channels (PTC, SC and MTC with different numbers of channels).

number of mappings that can be created and deleted within a transmission interval. Thus, the SC channel scales better in SMP setups, while still achieving the same rate of 500bits/s as the PTC in UP setups.

Since the amount of data that can be transmitted per time interval determines the maximum bit rate, we introduced multi-bit transmission (Section 5.5). The MTC (Section 5.4.3) leverages this multi-bit transmission by opening up multiple sub-channels. This optimization along with the MTC's distinguished reliance on a different Fiasco.OC mechanism for transmission makes it the fastest of our clocked channels. Tab. 5.1 shows that we can achieve a maximum bit rate of 2000bits/s when employing eight sub-channels.

5.7.2 Self-synchronized Transmission

In this section, we examine the self-synchronizing transmission method (Section 5.6). All tests use the MTC for both synchronization and data transfer. The results (Tab. 5.2) show that the channel capacity, of course, grows with the number of channels. But the channel capacity does not scale linearly, because transmitting a single bit requires either two or three operations, depending on the value to be transmitted. Moreover, the synchronization overhead for two transmission steps is five operations.
We also observed that the channel capacities scale much poorer with the number of CPU cores used. We attribute this to resource conflicts in the memory subsystem (cache, memory bandwidth) as each operation has to scan through a 16KBytes mapping tree. Still with the self-synchronized transmission, leveraging both CPU cores, we can achieve a bandwidth of $\sim$12kbits/s.

5.7.3 Impact of System Load

We performed all previous experiments on a system without any additional load. As real-world systems are usually not idling all the time, the question arises as to how load on the system impacts the throughput of the covert channels. To answer

CPU cores (#)	Nr. of channels (#)	Throughput (bits/s)	Gain (%)
1	1	3511	-
	2	5408	54
	4	7449	38
	8	9207	24
	16	10457	14
2	1	5605	-
	2	8782	51
	4	12166	43

Tab. 5.2: Throughput depending on the number of transmission channels.

System load (%)	Throughput (bits/s)
95	455
75	2292
50	4589
25	6892
5	8742
0	9207

Tab. 5.3: Throughput under load. Self-synchronized transmission with the MTC (8 channels). Sender, receiver, and the additional load all run on the same CPU core.

this question, we designed an experiment where we run a process that generates additional load and measure the channel throughput under the given circumstances. For all experiments with system load, we used the self-synchronized transmission with the MTC (8 channels). We used Fiasco.OC with the fixed-priority scheduler, so that load could be easily generated by a highly-prioritized thread that alternates between busy looping and sleeping. We also pinned sender, receiver, and the additional load all to the same CPU. We verified the correctness of this behavior by reading this thread's execution time, an information provided by Fiasco.OC.
As the results in Tab. 5.3 show, the achievable throughput is directly proportional to the CPU time available to the conspiring agents. In keeping with the expectation for self-synchronized transfers, all data arrived unscrambled.

Part III

Defenses

6 Uncovering Mobile Rootkits in Raw Memory

The majority of security solutions for ARM-based devices are tailored to the mobile market and focus on the detection of rather unsophisticated application-based malware. But the risk of such a device being used in a targeted attack can not be dismissed. Besides, the fact that a large fraction of ARM-based devices run a Linux kernel (the same kernel that is running on many desktop computers) renders them just as vulnerable to rootkits [18]. Even worse adversaries can choose from an already existing arsenal of attack vectors [32].

To counter the threat of a rootkit infection there exist a lot of application-based rootkit detectors [82, 19, 120, 105, 68]. Even though some of the detectors cloak their presence in the system, they still run with the same privileges as the rootkit itself, and therefore might be disabled by a sophisticated one. So, under the assumption that an adversary succeeds in implanting a kernel rootkit into a system, the chances of reliably detecting and removing it in a conventional – that is, non-virtualized – system are low. The underlying fundamental reason is that within a monolithic kernel, modularization is by convention only. There are no hardware mechanisms that would hinder a determined adversary with sufficient knowledge to arbitrarily modify kernel code and data structures with the goal of thwarting any detection and removal attempt.

To overcome the problem to reliably detect a rootkit without the rootkit having the chance to interfere with the detector, earlier attempts on the x86 architecture utilized virtualization technology. VMI is a concept first proposed by Garfinkel et al. [49] in 2003, where a dedicated detector VM runs side-by-side with a host VM that might be infected with a rootkit. The HV exports the system state of the host VM to the detector VM, who can then check this system state for discrepancies. A major obstacle however remained with this approach. Unlike a detector that directly runs *inside* the kernel, this *external* detector VM has to reconstruct the system state of the host VM from raw memory. This means the detector VM has to overcome the *semantic gap*. A solution was proposed in the year 2007 by Jiang et al. [62], who introduced the term semantic view reconstruction. The authors showed several techniques how to reconstruct kernel data structures from raw memory. Over the

years the concept evolved and even tools like `ps` or `lsmod` were recreated to provide the according information only based on a raw memory snapshot [56].

Due to concerns over the complexity of the resulting system architecture the concept was not yet considered on mobile devices. But in this chapter we show that a lightweight rootkit detector can be constructed with an off-the-shelf ARM-based device utilizing the ARM VE. The advantage of the resulting architecture is twofold. First, by running the rootkit detector in a dedicated VM, we make sure that an adversary cannot outright disable it. Second, the virtualization layer provides a snapshotting mechanism whereby the detector can capture the complete, untainted state of the host VM, comprising the architectural registers and physical memory, at a given time and run extensive analyses on it without having to halt the VM. Additionally, the snapshotting mechanism is generic in that a detector can strike the balance between thoroughness and runtime overhead that is best suited for its use case.

6.1 Mobile Rootkits

The term rootkit as defined by Hoglund [58] is a *kit* consisting of small programs that allow an adversary to gain (and maintain) root, the most powerful user in a system. But over the years rootkits evolved. Nowadays, instead of maintaining root, most rootkits directly reside in the OS kernel. The infection phase then involves loading the rootkit into the said. Traditionally, adversaries used LKMs (Loadable Kernel Module) [81] on Linux to infect the system. But loading a LKM requires the adversary to have root privileges in the system. Therefore, adversaries try to exploit vulnerabilities in applications or services on the system that run with root privileges to abuse those privileges to install the rootkit. The drawback of this approach is that it leaves footprints in the system (i.e., the kernel module containing the rootkit), thus exposing itself to detectors. As a consequence, rootkits started to adopt techniques as proposed in [95, 107, 119] to modify data structures in kernel memory directly via interfaces like `/dev/mem` and `/dev/kmem` or to simply hook existing LKMs. Once the OS is successfully infected, the rootkit serves as a stepping stone for future attacks.

Tab. 6.1 gives an overview of rootkits for the Linux kernel. Since rootkits often manipulate very low-level data structures of the kernel, most of them are hardware platform dependent. We list only rootkits that work on the ARM architecture.

We selected a number of exemplary rootkits from different categories. Cloaker [38] and CacheKit [122] are two academic rootkit that leverage novel mechanisms to

Name	Module loading	Module hiding	Arch. state manipulation	Use raw sockets	Process hiding	Syscall table manipulation
Cloaker (PoC)			X			
CacheKit			X			
Phrack issue 58						X
Phrack issue 61	X				X	
Phrack issue 68			X			X
Suterusu	X				X	
XOR.DDoS	X	X		X	X	

Tab. 6.1: List of existing rootkits that target ARM-based devices and the features they use.

hide itself from detectors. Both use architectural features of the ARM processor architecture to make a detection harder. We also list the concepts described in the Phrack magazine [95, 107, 119]. Even though these are not rootkits for themself, but they describe mechanisms that rootkits have adopted. Finally, we list two real world rootkits, XOR.DDoS [109] and Suterusu [81]. XOR.DDoS is a sophisticated binary only malware which was first spotted in the wild in September 2014. It consists of a malware core with an optional rootkit component. It tries to determine the Linux kernel version it is currently running on. This information is transmitted to a command-and-control server, which then tries to build a module for this specific kernel. If it is successful, the compiled module is sent back and loaded into the victims kernel; otherwise XOR.DDoS operates just as user space malware. If XOR.DDoS is able to inject the module, it provides the classical rootkit services (process hiding, file/directory hiding, LKM hiding, etc.). Suterusu on the other hand is an open-source rootkit that works on a variety of processor architectures (ARM, x86, x86-64) and provides services similar to the one provided by XOR.DDoS.

6.2 System Architecture

Before we present our rootkit detector in the next section, we will describe our system architecture. Our architecture is based on a Type-I HV called Perikles which was developed at the chair of SecT (TU Berlin). We have extended it with a mechanism whereby a VM can take a snapshot of another VM, comprising both (guest physical) memory as well as the full architectural register state. The architecture is depicted in Fig. 6.1. We run two VMs under control of the Perikles HV: the host VM, a full Android stack, and the detector VM, a minimal Linux with a special cross-VM inspection driver and our inspection tools. The detector VM can initiate a state snapshot of

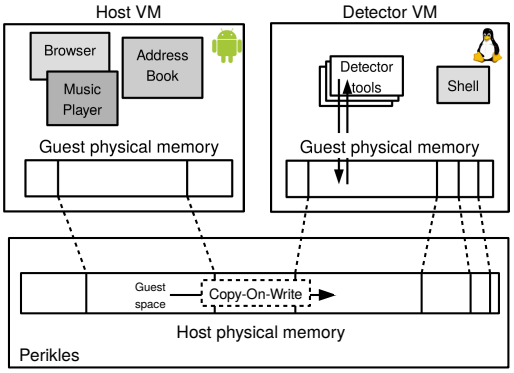


Fig. 6.1: System architecture.

the host VM, which is stored in a dedicated memory region, the snapshot space. The snapshot buffer appears as a special (guest physical) memory region in the detector VM, from where a kernel driver exports it via a device in the devfs. The detector runs as a user-space process and can perform the usual operations on the device (`open`, `read`, `seek`, etc.) Note that the right to take a snapshot does not entail the privilege to change the state of the host VM.

Since taking a memory snapshot involves copying large amounts of data, we use a *copy-on-write* (COW) mechanism to incrementally copy the entire memory of the host VM. That way, the operation of the host VM is only slightly slowed, but not suspended for an appreciable duration. Our implementation leverages the fact that the stage 2 page table, can specify access rights.

To initiate a snapshot, the access rights in the stage 2 page table of the host VM are set to *read only* while it keeps executing. Whenever a (guest) physical page for which no snapshot copy is taken yet is modified, an abort is raised in the Perikles HV, which makes a copy and sets the stage 2 permissions back to *read-write*. In addition the detector VM can also issue a hypercall to copy specific pages to the snapshot space or enforce the completion of a consistent snapshot. This is because relying only on the COW mechanism to finish the snapshot might take a while. Both operations are executed in the context of the detector VM and accounted to its processing time by the system scheduler.

In addition to the memory snapshot, we also take a snapshot of the architecture state. This gives us access to control registers of the target which are crucial in reconstructing the system state, e.g. `TTBCR`, `TTBR0`, and `SCTLR`.

6.3 Rootkit Detector

The snapshotting mechanism described in Section 6.2 alone is not capable of detecting a rootkit by itself. In this section, we will describe the detector we have developed.

6.3.1 Checking the Kernel's Integrity

Once a rootkits managed to exploit a vulnerable kernel interface, it starts manipulating kernel data structures to hide its presence and also to stay in control. In this section we examine how they specifically do this when loaded into the Linux kernel on the ARMv7 architecture.

Syscall table Processes can request services from the Linux kernel, via the kernel's syscall interface. The process loads a value into a designated register[1] before issuing the supervisor call instruction which traps into the kernel. The value in the register is then used as an index into the syscall table, which holds function pointers to the kernel functions implementing the requested service. Modifying the syscall table is thus a convenient target for a rootkit because it allows to hide malicious code, potentially without disrupting normal functionality. But in order to overwrite entries in the syscall table the rootkits faces the challenge of finding the location of the syscall table in memory. As opposed to some older versions of the Linux kernel, current versions used for Android do not export the `sys_call_table` symbol through the file `/proc/kallsyms`. There are still two ways to obtain the location of the syscall table. Either the adversary has access to the exact kernel image that is running on the device at hand, then he can retrieve the location from the image and program it directly into his rootkit. The other option is obtaining the address during runtime as described by [119, 54]. On ARM, every syscall enters the kernel via a software interrupt (SWI). SWIs are routed from the user to the kernel via the vectors page. The actual handler for SWIs is located at a fixed location (`0xffff0420`) Inspecting the follow up memory and identifying the case where a specific syscall is handled (e.g., the `sys_fork` syscall) allows the rootkit to calculate the base address of the syscall table. To detect rootkits that manipulate the syscall table our detector stores a hash of the initial syscall table in the detector VM and then periodically computes a new hash from the snapshot memory, and compares it with the initial one. As we exactly know which Linux kernel image runs in the host VM we can obtain the location of the syscall table in the same way as a rootkit would have to obtain it (by analyzing the kernel image).

[1] The ARM EABI uses the register `r7`.

Vectors page Apart from the syscall table adversaries often target the vectors page. Every transition from PL0 to PL1 diverts the flow of execution to the vectors page (see Section 2.1) and overwriting one of the vectors gives the rootkit control on every exception of that type. An academic rootkit by David et al. [38] even relocates the original vectors page to a different memory location and places its own copy. Therefore, we do not only check the actual memory location of the vectors page but also the registers that control its location (`SCTLR` and `VBAR`). Luckily, Linux sets the vectors page to a fixed location (see Section 2.1) and never changes it during runtime. For our detector to uncover such a rootkit, we therefore not only validate the content of the vectors page. We also validate its location by taking a snapshot of all privileged architectural registers, which include `SCTLR` and `VBAR`.

Arbitrary code changes Finally, the rootkit could opt to overwrite selected functions in the kernel so as to redirect the control flow. Such manipulations can be detected by computing a hash over the kernel's text section and comparing it to a pre-computed one. The only issue that remains is, in certain configurations, the Linux kernel patches its text section during startup as part of its normal operation. This renders precomputed hashes of the text section useless. But, under the assumption that during boot, the kernel is still unmanipulated (e.g., the bootloader could check its integrity), we trust the kernel in this early stage to send a notification to the detector after it has set up itself. The detector then computes a checksum over the text section, which is later used for comparison.

6.3.2 Reconstructing Hidden Kernel Objects

Aside from detecting the rootkit itself as described in the previous section, undoing the cloaking that is performed by the rootkit might be another goal of a detector. In this section, we describe which hidden objects can be recovered by our detector.

Modules A common way whereby rootkits infect the kernel is loading a kernel module. Naturally, they seek to hide the module's presence afterwards. The Linux kernel exports the list of modules through the subfolder `/sys/modules`. Tools such as `lsmod` then use this sysfs directory to display the loaded modules in a human-readable representation. Now to hide its presence a rootkit tries to overwrite entries in the `inode_operations` structure belonging to the module `sysfs_dirent` structure of the `/sys/modules` folder. To uncover such manipulations, our detector performs three steps. First, it searches through the memory snapshot based on specific patterns to identify all modules in the `/sys/module` folder. Second, it iterate over the memory snapshot again and searches for a pattern that matches the module data

structure. Finally, it verifies that the inode operations for the module are correct and have not been overwritten.

Processes To generate the list of currently running applications, tools like `ps` or `top` read the procfs top level directory. This directory again contains directories named by the PID of all currently running processes. Now, some Linux rootkits, such as [119, 106, 81], modify the functionality of the proc filesystem (procfs) to hide selected processes. To that end, they usually overwrite the `file_operations.read()` function of the procfs top level directory. Because, once an application reads the top level directory of the procfs the list of running processes is created on the fly. The read function in the kernel iterates through the list of `task_struct` and creates the directories of the procfs top level directory dynamically. The rootkit only minimally changes the functionality. It just skips over some `task_struct` such, that their directories in the procfs top level directory are not created. But processes can also be identified without directly iterating through the `task_struct`. Instead, our detector works as follows: Each process contains a kernel stack. These stacks are aligned to 2KBytes. On a specific offset of that kernel stack is the process' `thread_info` structure located. This `thread_info` structure has a pointer to its corresponding `task_struct` structure. The `task_struct` structure in turn contains a pointer back to the beginning of the process' stack. Our detector iterates through the snapshot memory in 2KBytes increments, and tries to find such relations. Once it found a matching pattern we use the identified `task_struct` structure to extract all required information (e.g. PID, name of the process, etc.). We compare this manually created list with the one provided from within the host VM, if there are any discrepancies we identified a hidden process (and a potential rootkit infection).

Sockets Given the goal of long-term intelligence gathering, an advanced rootkit is likely to communicate with an outside party over a network socket. The rootkit wants to hide this socket as well. While reconstructing the process list from the structure `task_struct`, we are also able to identify open files and more important open sockets, which are always associated with processes. Every process has a member called `files_struct *files`, which represents the list of files (and sockets) associated with this process. By looking up the kernel `socket_file_ops` structure and comparing it with the actual `f_ops` structure of the currently investigated file, we are able to determine whether the handle is for a file or a socket connection. If the entry is a socket, the socket structure provides us with information about the network connection. Again the list of open sockets is compared with the one from the procfs directory (using the tool `netstat`). If we identify any discrepancies between the two lists, we have uncovered a hidden network connection (and a potential rootkit infection).

Operation	Rootkit		Detector
	Suterusu	PoC	
vector page manipulation			x
vector page relocation			x
syscall table manipulation		x	x
function hooking	x		x
function pointer manipulation	x		(x)

Tab. 6.2: A list of common manipulations. The Suterusu rootkit and a PoC rootkit perform a number of manipulations. Our detector is capable to detect each one.

Files Typically, rootkits serve as a means of hiding the infiltration of a system and ensure its persistence. Often, further activities require data to be deposited in the file system in an undetectable way. Instead of changing the entries for the relevant syscalls – which is easily detected –, an adversary may choose to replace function pointers in `struct file_operations` pointed to by `struct file` [81]. For the adversary, this approach has the advantage that file objects are dynamically allocated, which makes their detection more complicated.

6.4 Evaluation

The evaluation section is split into three parts. First we evaluated how reliably our detector can detect a rootkit. In the second part (Section 6.4.2), we measured the time the detector needs for a reconstruction of specific elements. We performed multiple Android benchmarks as well as LMBench to measure the virtualization overhead and the overhead induced by the snapshotting mechanism (Section 6.4.3). All experiments were conducted on a Cubietruck (Allwinner A20, 2x1.06GHz CPU, 2GB RAM) running Android 4.4.2 on a Linux 3.4.0 kernel.

6.4.1 Detector Efficacy

To test the efficacy of our solution, we have tested it against two exemplary rootkits: Suterusu [81] and a nameless proof of concept rootkit [40]. The results are shown in Tab. 6.2 and Tab. 6.3.

Our detector picks up manipulation to and relocation of the vectors page. Since neither of the two specimen under test manipulates the vectors page. We tested this ability with a small extension to Suterusu. Also, changes to the syscall table are detected. Function hooking, overwriting function code, causes changes in the checksum of the kernel text section, which our detector notices. As yet, we do not check the integrity of genuine kernel modules, though. Unlike the processes

Operation	Rootkit		Detector
	Suterusu	PoC	
module hiding	x	x	x
process hiding	x		x
connection hiding	x		x
file hiding	x	x	

Tab. 6.3: Object reconstruction.

and connections, for which the underlying kernel data structures are guaranteed to be memory resident, file-associated data structures may or may not reside in memory. As such, our detector cannot reconstruct them from a snapshot of the guest's physical memory.

6.4.2 Kernel Object Reconstruction

Tab. 6.4 gives a short description for each tool we used in our analyses. We describe its purpose and examined property and list the required runtime to extract the respective properties. The reconstruction of some kernel structures is rather costly because we have to iterate through the memory snapshot multiple times.

Tool	Description	Time (in sec.)
gsnps_procfs	Check procfs `fops`	0.3790
gsnps_proc	Extract `task_struct` process list	0.1350
gsnps_sysfs	Extract sysfs module list	20.1980
gsnps_mod	Extract `module` structures	17.3520
gsnps_sock	Extract `socket` list	0.2130
gsnps_exec	Hash kernel `.text` section	0.5689

Tab. 6.4: Runtime of tools to extract specific information from the memory snapshot.

To check the integrity of the kernel text section, we used the mbed TLS library [11] and computed a SHA1 hash over the text section.

6.4.3 Application Benchmarks

We evaluated our snappshotting mechanism with a number of performance benchmarking suits. We used the well established LMBench (v3) suite for Linux and the two Android benchmarking suites Antutu (v5.7) and Geekbench (v3.3.2).
All benchmark results can be obtained from Tab. 6.5. As for LMBench, we ran a number of relevant latency and bandwidth benchmarks. In most of the benchmarks, the virtualized setups show only slight performance degradations. The bandwidth benchmark showed good results, during the time of taking a snapshot. This is due to the fact that the majority of copy operations is performed by the copy thread on

the secondary CPU core, and not as a reaction to a COW-incurred page fault on the host VM. Though, the host VM still has to trap into the HV to flush the TLB, which explains the small, but noticeable increase in execution time.
The Antutu benchmark results revealed that the snapshotting only has a small impact on all but I/O intensive apps. Taking a snapshot incurs a notable impact of $\sim 20\%$ on the *RAM Speed* benchmark. This is not surprising as this benchmark excessively accesses main memory which then collides with the main memory accesses originating from the snapshotting operations.
On the other benchmarks the snapshotting showed only a marginal performance hit. The results show that the performance is almost on par with the HV measurements taken when no snapshot operation is in progress. Worth mentioning is also that the memory footprint of the Android system is quite large. After performing a single Antutu benchmark run, $\sim$75% of the memory pages were copied due to COW alone.
The results for Geekbench can be obtained from Tab. 6.5. The results of the scenarios *Perikles + Android (nosmp, mem=768)* and *Perikles + Android (snapshot)* again show a $\sim$3% performance penalty due to the virtualization. Apart from that, the numbers are in line with the expected results.

	Operation	Android (smp, mem=2048m)	Android (nosmp, mem=768m)	Perikles + Android (nosmp, mem=768m)	Perikles + Android (snapshot)
LMBench	`lat open syscall`	15.92	16.84	16.94	17.01
	`lat read syscall`	0.58	0.58	0.62	0.64
	`lat write syscall`	1.74	1.75	1.89	1.89
	`bw rdwr`	666.92	590.61	560.74	540.96
	`bw frd`	822.7	732.82	730.6	715.47
	`bw fwr`	277.2	279.64	283.4	256.13
	`bw fcp`	783.94	651.4	639.08	632.36
	`bw cp`	275.83	250.31	251.65	222.4
Antutu	Overall	9888.4	7873.0	7624.0	7728.6
	Multitasking	1580.4	780.0	802.5	814.6
	Runtime	650.2	357.5	363.0	353.2
	CPU Integer	576.0	283.5	290.0	293.0
	CPU Floating-point	593.2	291.75	299.5	307.6
	Single thread integer	818.4	785.5	795.75	804.8
	Single thread floating-point	674.4	643.75	653.75	660.4
	RAM operation	494.8	247.75	248.5	253.0
	RAM speed	807.2	785.0	705.0	639.2
Geekbench	UMP stream copy	1300.0	1232.0	1202.0	1220.0
	SMP stream copy	1302.0	1248.0	1220.0	1230.0
	UMP stream scale	870.84	816.58	818.9	818.78
	SMP stream scale	1184.0	872.82	847.26	838.56
	UMP stream add	420.66	434.74	439.78	436.72
	SMP stream add	734.74	455.88	456.18	450.16
	UMP stream triad	410.3	436.86	431.44	425.48
	SMP stream triad	671.98	445.92	438.22	440.56

Tab. 6.5: Benchmark results for the different system configurations

7 Hypervisor-based Execution Prevention

Mobile devices have become versatile, because of their move away from proprietary OSs towards an open system architecture. The downside of this development is the dramatic growth in software complexity. As with desktop computers and servers before, this complexity renders mobile devices vulnerable to malware.

As is to be expected in an arms race, the complexity of cyber-attacks increases constantly, with rootkits as particular menacing. By undermining the OS, kernel rootkits subvert the system in such a way that they are in position to potentially disable any countermeasure taken against them. The reason behind that lies in the monolithic structure of all currently used mainstream OS kernels, which cannot afford address space protection for kernel subsystems. Once a rootkit has penetrated the kernel, no anti-malware measure can trust kernel services, which it needs to function. As efforts to reduce or even weed out kernel vulnerabilities are unlikely to succeed for the foreseeable future, architectural changes are the only effective counter-strategy. Indeed, virtualization-based approaches [90, 97] have shown promise as rootkit mitigation.

In this chapter, we present an extension to the Perikles HV that effectively blocks attack vectors commonly used by kernel rootkits: unapproved kernel code (either injected or modified) and execution of user code with kernel privileges.

7.1 Assumptions and Threat Model

In this section, we discuss the assumptions we make regarding the initial system state. Afterwards, we present our threat model and the defense capabilities provided by our mechanism.

7.1.1 Assumptions

We assume a trusted system boot procedure that ensures the authenticity of the HV, the system configuration description, all guest boot images including accompanying components such as initial ramdisks. For the guests, we assume a standard boot sequence: after the boot image and the ramdisk are loaded, control is transferred to an entry point along with boot parameters. At this point paging may not be activated within the VM. The guest decompresses the kernel, relocates it, creates PT for the decompressed kernel image, and activates paging before calling the kernel entry point. From this moment on, the location and size of the different kernel regions (`.text`, `.rodata`, and `.data`) are known and fixed. Furthermore, the code executed before paging is enabled depends only on the boot control arguments. These are passed to the kernel by the bootloader through the atags structure or a device tree. We therefore argue that these data structures along with the code that parses them can be considered safe and uncompromised, their integrity being guaranteed by one of the *trusted boot* mechanisms [4, 104, 42].

7.1.2 Considered Attacks

We consider two attack vectors, the *return-to-user* (ret2usr) [65], and the more recent *return-to-direct-mapped memory* (ret2dir) [64]. Both are advanced code injection attacks, which aim at the manipulation of a kernel pointer to gain control over the flow of execution. Depending on the attack, the shellcode is placed in user or in kernel memory. In the classical ret2usr attack, an adversary tries to jump to code in the user memory (Fig. 7.1 ①). However, advances in processor architectures have brought protection against this type of attack (Fig. 7.1 ②). On ARM the feature is called Privileged Execute Never (PXN) [6]. In a ret2dir attack, instead of jumping to shellcode placed in user memory, the adversary jumps to code in the kernel memory (Fig. 7.1 ③). The adversary exploits the fact that the Linux kernel has the physical memory mapped into its own virtual address space (called `physmap`). Shellcode placed in user memory is thus also accessible through a kernel alias. The adversary avoids the jump into user memory, circumventing the existing countermeasures.

7.1.3 Threat Model

The guest OS kernel is considered trusted only in the early system bootup phase and untrusted afterwards. We do not expect it to withstand attacks that aim at gaining full control of the guest kernel. Under these conditions PT-based security mechanisms (Non-executable, read-only) cannot be relied on. We assume an adversary will attempt to install function hooks in the kernel to gain control in opportune moments.

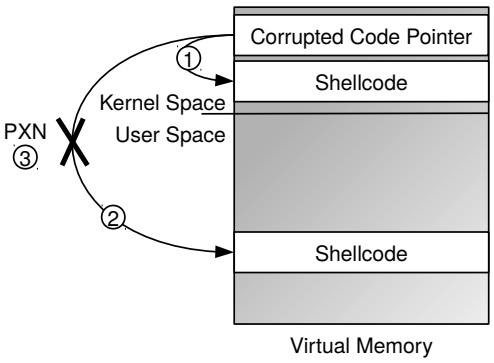


Fig. 7.1: Operation of ret2dir and ret2usr attacks on the Linux kernel.

For that, he will either overwrite existing kernel code or change function pointers so that they point either to injected kernel code or user code. We note that such a need exists in most kernel rootkits today.

We do not consider control flow manipulation attacks. This means that it is possible for an adversary to launch a ROP [98] attack by using only existing kernel code snippets. However, for that matter we refer to the fact that without its own kernel code, this type of attacks tends to have limited functionality. Any attempt to introduce own code into the kernel should be defeated by our mechanism Additionally, orthogonal security solutions exist for protecting control flow integrity [113].

Also, we do not consider attacks whereby devices are used to overwrite critical memory regions (e.g., DMA attacks [17, 102, 20]). The issue of secure device virtualization is orthogonal to the examined problem of rootkit defense.

7.2 Execution Prevention

In this section, we discuss the four design goals we have specified for an execution prevention mechanism. Then, we present our formal definition and the requirements that have to be fulfilled in order to achieve the desired security properties. Finally, we describe how our execution prevention uses existing hardware protection mechanisms to achieve effective privileged execution prevention for unapproved code.

7.2.1 Design Goals

In addition to our main goal – allowing only authorized code to be executed with kernel privileges – we had four design goals:

Small size For security sensitive systems, it is crucial that their TCB is as small as possible. The rational behind this requirement is that only with small code sizes it will be possible to thoroughly audit the source code in order to gain confidence in its correctness. Prospectively, using formal methods also place limits as to the size of the examined source code.
This requirement automatically applies to the most privileged layer in the system architecture, in our case the HV. It follows that the number of services offered is small, smaller than those offered by other solutions with a focus on rich functionality.

Minimal guest changes Adding new mechanism to a system might require changes to the guest OS. We aimed at keeping the modification needed for guest systems as low as possible. Full (CPU) para-virtualization was dismissed as too intrusive. Conveniently, most current ARM processors support VE so that hardware-supported virtualization was chosen as a starting point.

Good performance We set the goal to keep the incurred runtime penalty as small as possible. It was deemed acceptable to employ para-virtualization for selected functionality if that helps to cut down on a performance bottleneck.

Scalable architecture The mechanism aims to be compatible with a range of system architectures, including those that structure the system by leveraging multiple VMs. This goal of scalability rules out solutions that are limited in the number of supported domains, such as TZ.

7.2.2 Definition

Linux enforces the access permissions in the following way. While running in PL0, applications can access (read/write) data that is part of their user memory. Of course they cannot read/write memory belonging to the kernel. On the other hand, the kernel needs access to user memory for operations such as `copy_from_user` or `copy_to_user`. Looking at the execution permissions reveals the same picture. Ordinary applications are not able to execute arbitrary kernel code. Again, the contrary is not true. While running in PL1, kernel code (e.g. a driver) can execute code residing in user memory. Thus, a malicious kernel subsystem can manipulate the execution flow to execute code from user memory. However, unlike the access permissions, which are required by the Linux system, executing user code while running in PL1 is never necessary and should be prevented.

Our execution prevention ensures that the execution permissions are confined. While executing in PL1, no user code can be executed. Furthermore, to prevent all attacks

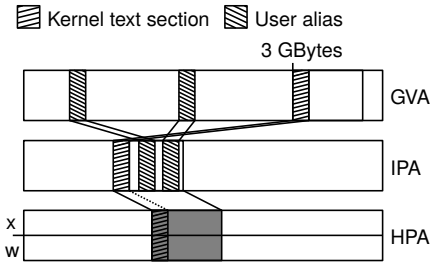
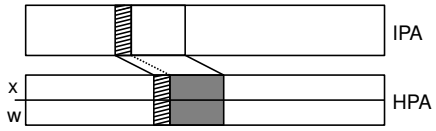
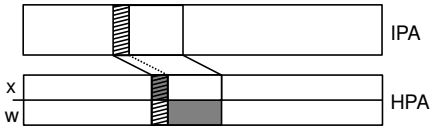


Fig. 7.2: A generic stage 2 PT memory layout. All entries are writable and executable.



(a) `WX` permissions enforced by the stage 2 PT for PL0.



(b) `WX` permissions enforced by the stage 2 PT for PL1.

Fig. 7.3: The two different stage 2 PT layouts as enforced by our execution prevention.

described in Section 7.1, the enforced permissions are even more restrictive. Only parts of the kernel memory that hold genuine kernel code are executable. However, enforcing this system behaviour by a higher privileged entity requires tracking the memory layout of the guest OS. In the general case this is difficult, because the HV cannot make any assumptions about the memory layout of the guest OS. Therefore usually all pages in the stage 2 PT are writable and executable (Fig. 7.2). The hatched segment shows the `.text` section of the Linux kernel.
With our execution prevention mechanism Perikles imposes restrictions on executable regions depending on the PL. We have created two memory layouts, enforced by the stage 2 PT as illustrated in Fig. 7.3a for PL0 and in Fig. 7.3b for PL1. Fig. 7.3a shows that only the user memory is marked as executable and writable in the stage 2 PT, whereas kernel memory is not. This is a property already guaranteed by the Linux kernel. However, to rule out implementation bugs this is additionally enforced by our mechanism. A more critical situation arises when the system executes in PL1. As illustrated by Fig. 7.3b, the kernel's text section is executable and not writable while everything else is writable (to support `copy_from_user` and `copy_to_user`) but not executable (to rule out ret2usr and ret2dir attacks).

7.3 Implementation

In the following section we discuss the implementation of our design. Our prototype was build for the Cubieboard 2.

7.3.1 XN Enforcement

ARMs stage 2 PT format contains an `XN` bit, which renders pages non-executable when set. In order to enforce the previously defined properties, the PT entries must have set this bit on different memory regions depending on the PL. There are two ways to implement this. We can either rewrite the PT to mark the kernel memory as non-executable when executing in PL0 and vice versa when executing in PL1. This has however serious performance implications. The other approach is to maintain two distinct PT, one for PL0 and one for PL1. The increased memory footprint to store two stage 2 PTs for every VM is a trade off we make for better performance.

Upon bootup, the memory that must be executable in PL1 encompasses two regions. The first memory region is the exception vectors page. The address of the exception vectors can be inferred by the HV by reading the `VBAR` and `SCTLR` registers of the guest. The exception vectors page is 4KBytes in size. The second region is the `.text` section of the Linux kernel. The start and size of the `.text` section depends on the number of features built into the kernel. It can be extracted from the Linux kernel binary. Defining the memory region that must be executable while in PL0 only requires the HV to know what virtual memory split Linux is using. The split can be obtained from the Linux `.config` file or also from the Linux kernel binary. The addresses in the binary are already virtual addresses and depending on the memory split, start at `0x40000000`, `0x80000000`, or `0xc0000000`. Everything below this start address is user memory.

When setting up the system to run with two sets of PT, each transition from PL0 to PL1 or from PL1 to PL0 involves a trap into the HV. This is because transitions from PL0 to PL1 use the exception vectors page, but the exception vectors page is not in the executable set of PL0, thus generating a prefetch abort. The HV then checks the address where the fault occurred and the PL at the time of the fault. Based on this information, the HV decides whether the guest user application performed a valid kernel entry. If the PL matches the address the guest tried to execute from, the HV loads stage 2 PT for PL1 and resumes the execution of the guest. Then transition in the other direction (from PL1 to PL0) works in the same way. When the guest system executes in PL1 and wants to exit the kernel to continue the execution of a user application, the attempt to execute the first PL0 instruction also generates a prefetch abort which again traps into PL2. The HV then again checks the address that caused the fault and the PL. If both match the HV switches to the stage 2 PT of PL0 and resumes the execution of the guest.

7.3.2 TLB Management

The TLB caches PT translations to avoid costly PT walks for every memory access. As the MMU fetches PT entries when needed, adding rights to a PT entry does not need further attention. But when the rights of a PT entry get reduced, the TLB needs to be informed so that an old cached entry gets purged. This means it is the obligation of the OS kernel to keep the TLB and the active PT consistent.

As loading PT entries into the TLB is expensive, it is desirable to avoid reloading as much as possible. However, it has to be ensured that only entries belonging to the currently active context are used to translate virtual addresses. In order to mitigate the impact of frequent MMU switches on the TLB, the ARM architecture includes an ASID into each entry. A memory context is identified through a PT base address and an ASID, both of which are held in a register, the `TTBR`. Only entries whose ASID matches the currently valid ASID are used for translations. The advantage of this scheme is that a memory context has its TLB entries retained in the TLB even over context switches. When the context gets reactivated, it finds its now again active entries in the TLB and does not need to reload them from slow memory.

With VE, ARM introduced another identifier, the VMID. In addition to the address of the currently active PT, the `VTTBR` also holds this VMID. Before the HV executes a guest, the `VTTBR` is loaded with the stage 2 PT associated with this guest along with a VMID assigned to that guest VM. As with ASIDs, by using multiple VMIDs, the TLB can hold entries of multiple VM contexts. For the guest, TLB maintenance operations, in particular TLB entry invalidations, work as in a non-virtualized environment. When the guest kernel executes such an operation (e.g. `TLBIALL` or `TLBIASID`) only entries from the currently active VMID are removed from the TLB.

To efficiently enforce different execution rights depending on the guest privilege level, it is opportune to use two different VMIDs. The translation from IPAs to HPAs is identical; however, the execution rights differ. With different VMIDs, it is possible to hold PL0 and PL1 TLB entries with the correct execution permission in parallel in the TLB. But now all TLB maintenance operations must be synchronized and executed for both VMIDs. However, guest TLB operations are limited to the current VMID, which leaves flushing guest PL0 TLB entries (which are tagged with a different VMID) to the HV.

To achieve the synchronization, we have investigate two strategies:

- **fully virtualized MMU (fvMMU)** The VE allow for trapping different instruction classes, among them TLB maintenance operations. On an intercept, the HV

evicts all TLB entries of the guest that match the invalidation criterion[1]. While this approach is transparent for the guest, it is also slow because each TLB invalidation traps into the HV. Worse yet, if an entry associated with PL0 has to be evicted, two slow VMID reloads are necessary. The HV has to activate the PL0 VMID, evict the TLB entry, and switch back to the PL1 VMID, because VMID-selective TLB invalidations are not supported.

- **para-virtualized MMU (pvMMU)** Instead of having all TLB operations trap, the intercept is disabled and the guest kernel is modified to cooperate with the HV. PL1 TLB flushes do not need HV support and are executed directly. PL0 TLB flushes are recorded on a page registered with the HV. As the HV gains control on the next transition from PL1 to PL0, it processes all queued TLB operations with the VMID set to PL0. These batched TLB updates cut down on the number of both entries into the HV and VMID changes.

7.4 Evaluation

Our extension to the Perikles HV that enforces the previously described properties is implemented in 280SLOC. For the pvMMU implementation we generated a patch for the Linux kernel, which consists of another 100SLOC.

To evaluate the feasibility of our approach, we ran a number of benchmarks on the Cubieboard 2 [31]. We conducted the experiments on the Perikles HV with a VM that runs Linux kernel version 3.4.90-r1. We disabled all power management or frequency scaling features in the guest VM (e.g. `CONFIG_CPU_FREQ` and `CONFIG_PM_RUNTIME`, etc.). Across all scenarios, we set the Linux preemption model to `Preemptible Kernel (Low-Latency Desktop)`.

7.4.1 Low-level Benchmarks

The results of LMBench show that running a virtualized Linux on top of a HV does not induce too much overhead. The entire set of results can be obtained from Table 7.1. The first column contains the results from a native Linux. The second column contains the results of an unmodified Linux kernel running on Perikles without the PT protection. The two remaining columns contain the results of a system running with the execution prevention, with the fvMMU and the pvMMU implementation as described in Section 7.3.2.

[1] Single virtual address, ASID or all

Benchmark	Native Linux	Perikles (no sep.)	Perikles (fvMMU)	Perikles (pvMMU)
`lat pipe`	19.68	22.37	45.35	34.03
`lat fork + exit`	765.37	1110.40	1526.50	1384.50
`lat fork + execve`	3095.00	4036.50	5135.00	5203.00
`lat select`	17.38	17.47	18.60	18.84
`lat read syscall`	0.66	0.67	2.79	1.95
`lat write syscall`	0.82	0.88	3.95	3.90
`lat open/close`	6.83	10.67	14.34	19.27
`bw mem rdwr`	813.77	733.80	729.58	729.36
`bw mem bcopy`	663.75	570.37	569.03	565.97
`bw pipe`	331.92	299.41	283.02	235.71
`bw unix sock`	274.15	251.13	224.46	225.57

Tab. 7.1: LMBench results on the Cubieboard2

The benchmark results show that different syscalls (e.g. read or write) show almost no overhead on a Linux system running on a HV. However, the PT protection induces some overhead due to the fact that on a syscall, the system traps into the HV to switch the PT. Thus, the additional latency comes from at least one HV roundtrip and an additional TLB operation. The other benchmarks (e.g. a `select` syscall) are only marginally slower than their native counterpart. This can simply be explained by the fact that they do not need a HV roundtrip. The application benchmarks in Section 7.4.2 show that the impact on real world scenarios is much lower.

In addition to the system level benchmarks, we performed the Integer Arithmetic performance benchmark *Dhrystone*. As we passed the FPU through to the guest VM without HV interception, we suspected almost no performance degradation in the virtualized setups. The benchmark was performed, as ARM suggests [41], in order to get meaningful results. Table 7.2 shows the results of the benchmark. Our assumption was right. Indeed, the *Dhrystone* benchmark shows similar results on all five system configurations.

Scenario	Dhrystones/sec (Average)	Dhrystones/sec (Standard deviation)	DMIPS
Native Linux	1266408.71	1135.71	721
Perikles no sep.	1264685.21	364.01	720
Perikles fvMMU	1264377.26	492.56	720
Perikles pvMMU	1264061.20	246.70	719

Tab. 7.2: Dhrystone benchmark on the Cubieboard2

7.4.2 Application Benchmarks

We performed two application benchmarks. We extracted a Linux kernel archive (`tar xJf linux-3.17.tar.xz`), and built a Linux kernel with `make allnoconfig && make`. The overall runtime of the operations was measured using the Linux tool `time`. The results can be obtained from Table 7.3. The archive extraction benchmarks

Scenario	Operation	
	Extract xz archive	**Build kernel**
Native Linux	7.38	11.74
Perikles no sep.	9.95	14.70
Perikles fvMMU	10.0	19.57
Perikles pvMMU	10.1	14.87

Tab. 7.3: Application benchmarks on the different scenarios (time in minutes - lower is better)

shows overall low overhead across all scenarios. The gap to Linux baseline is ∼26%. Among the virtualization solutions, the overhead is only ∼6% compared to Perikles without separation. This indicates that the slowdown is not primarily down to our implementation but can attributed to general virtualization overhead.

Taking Perikles without separation as a baseline and comparing it to the scenarios that are equipped with execution prevention shows that this feature only leads to a ∼1.5% performance degradation. Building a Linux kernel reveals that it is worth the effort to para-virtualize selected functionality – in our case the TLB operations – to increase the performance. This reduces the performance overhead compared to our baseline configuration from 33% (with the fvMMU) to ∼1.5% (with the pvMMU).

Part IV

Epilogue

8 Conclusions

In this thesis we demonstrated that applications should not rely on the isolation properties provided by commodity system software, when a high degree of security is demanded. These commodity systems lack proper isolation capabilities, due to complicated subsystems and complex kernel interfaces. We showed that both can be used to break the systems isolation and/or give an adversary full control over the system.
Instead, we postulated in this thesis that in domains where usability and security converge (e.g., mobile devices), system designers should leverage small statically partitioned HVs to properly isolate security or safety critical components from the rest of the system. Moreover, the HV's programming interface should be narrow, only exposing CPU-like interfaces. To accommodate for the coarse grained isolation that HVs provide, additional security components should be build into the HV in a modular and transparent way.
To demonstrate the feasibility of such an architecture we extended a small statically partitioned HV with two modular defense mechanisms that prevent or uncover attacks on the guest OS kernel. The first mechanism prevents code-reuse attacks by enforcing specific properties (non executable) on guest memory regions. The second mechanism provides the capability to snapshot a guest OS. The guest memory can then be inspected with a set of user space tools to uncover rootkits. Both mechanisms are designed in a modular way, only marginally increase the code base of the HV, and expose only a very simple interface.
Especially on mobile devices where the resource utilization is more predictable and on-demand starting and stopping of VMs is not a requirement our proposed architecture shows its strength.

In the remainder of the chapter we conclude each topic covered in this thesis individually. We recap on the design of the rootkit we presented in Chapter 4. We discuss the underlying issue that led to us being able to install our rootkit into the HV mode. Then we elaborate how the covert channels we found (Chapter 5) can be prevented. We again also analyse the underlying issue, that ultimately led to these covert channels in the first place. We conclude this chapter by evaluating our two defense mechanisms (discussed in Chapter 6 and 7). We briefly look into, how both mechanisms can be extended and how they could be combined to achieve a higher threat coverage.

Hardware Virtualization-assisted Rootkits In Chapter 4 we elaborated on the feasibility of gaining control over the VE and its highly privileged processor mode to install a rootkit into it. We showed the delicacy of the issue, by pointing out several attack vectors, whereby adversaries are able to subvert the OS kernel and install such a rootkit. Once installed, the rootkit is very hard to spot and to remove because it has full control over all system resources and can easily spy on the OS kernel as well as user applications. We implemented a full prototype rootkit, to demonstrate its feasibility. We evaluated our rootkit in terms of stealthiness and showed that many detection mechanisms (e.g. time drift, memory access times, etc.) would not work to detect it.

This issue raises the question whether we can deter an adversary from misusing the VE in the first place. It depends whether an already deployed system should be changed to lockdown the VE or if redeployment of the system software is an option to be able to make more profound changes.
If the user has full control over the platform, and the system firmware boots into the secure world, the secure world OS can prevent the hypervisor call instruction from getting enabled. Successive calls of the hypervisor call instructions would simply cause an exception. A more elaborate approach is to switch into the non-secure world in one of the earlier bootstages, giving Linux no chance to install its HV stub vectors, effectively locking down the VE.
The above fixes however require changes to the boot chain, which is usually under vendor control. Additionally, the early bootstages would already have to know whether VE lockdown is desired. This however would require an appropriate mechanism to signal whether to enable the VE or not, e.g., a runtime secure world service which irrevocably disables the VE until reset.
If the VE are still accessible when the bootstage leaves the secure world and the general purpose OS starts up, it is difficult to get them locked down. Moreover, it is remarkable that, there is no general mechanism to disable the VE other than disabling the hypervisor call instruction.

Disabling the hypervisor call instruction and not setting the HV's exception vectors to a valid address effectively disables the VE. But in scenarios where the users do not have access to one of the early bootstages to perform the needed steps, users can still try to make the VE unusable. These efforts are however flawed, and still run the risk of being subverted.
One approach is to use the Linux kernel's HV stub to set the HV's exception vectors to an invalid memory location. As there might be no way for the user to disable the hypervisor call instruction during runtime, the approach still leaves the adversary with the opportunity for a DoS attack, as executing the hypervisor call instruction then would lead to an endless exception loop.
An other approach comprises of installing a "nop" vector table, which just executes

the exception return instruction for every exception it receives. This solution however, suffers from the same problem as the KVM exception vector. The location containing these instructions is backed by physical memory. Yet, all physical memory is accessible to the OS, thus an adversary who managed to take over the OS kernel, could still find the location of this “nop” table, overwrite its entries, and gain control over the VE again.
To improve this effort the defender could create a stage 2 PT of his own to protect his “nop” vector table from being manipulated from code in the OS kernel. Accesses to this range would then either result in stage 2 page faults, which the ”nop” vector table could reflect back to the OS, or could be backed by invalid physical addresses or emulated so that the OS just sees invalid data. This solutions effectively leads to running a very small HV that just returns to the OS kernel once called.

In summary, most of our attack vectors exist because the user does not have control over all bootstages (see Section 4.2). On the Jacinto6 board TI installs its own secure world OS, and a kernel process can always call the secure monitor call instruction to request the installation of a HV. To prevent this attack vector, several approaches can be used. The non-secure world bootloader could call the secure monitor call instruction to install a lockdown HV as described above. Then the Linux kernel can run as usual but with a slightly reduced amount of memory. A few memory pages must be reserved for the stage 2 page table and for the stub HV code ($\sim$12KBytes).

The Linux HV stub was added to the kernel soon after Linux kernel release v3.6. Many Android devices still run kernel versions lower then v3.6 (e.g. v3.0 or v3.4). These devices then have a completely uninitialized PL2 mode. To prevent an adversary from exploiting this entry (see Section 4.2), an administrator or a user can seal PL2 as described for attack vector 1.

Breaking Isolation through Covert Channels In Chapter 5 we examined the Fiasco.OC microkernel. We depicted that it cannot deliver the promised isolation properties in the face of a determined adversary. We identified two shared facilities whose control mechanisms can be rendered ineffective (kernel allocator, object slabs) and a third, who lacks them entirely (mapping trees). In our experiments, we showed the feasibility of using them to form covert channels and achieved maximal channel capacities of up to $\sim$10000bits/s. Further experiments indicated that with additional refinements the achieved channel capacities could be more than doubled. Moreover, processing power increases, which accommodates the channel capacities. Therefore, to counter at least a few of our covert channels it seems advisable to introduce a mechanism whereby the switch rate between isolated domains can be controlled, like proposed by Wu et al. [115]. This would prevent our clock synchro-

nized channels to work or at least impact its bandwidth. Because for these channels to function correctly, both conspirators (sender and receiver) have to execute within a specific time frame one after the other. We however have to acknowledge that such a scheme may have a negative impact on system performance.

But apart from these very low-level mechanisms that allowed us to build our covert channels, there is also an issue with how Fiasco.OC encapsulates Linux and thus endows it with its full microkernel API. We acknowledge that there is no doubt about the need for the ability to encapsulate Linux instances. However, it seems ill-advised to offer Linux the full microkernel API, as the offered feature set is not fully needed, yet grows the attack surface.
This again reinvigorates the debate about whether system design shall be driven by pragmatism or principle. While proponents of Fiasco's pragmatism often point to the wide range of functionality it provides. Indeed, its multi-processor support, its ability to host Linux [69] on platforms without virtualization support, and the availability of a user-level framework [70] make for a system that lends itself to a wide range of applications.
In contrast, seL4 [67, 45, 66] is the first general purpose kernel for which a formal correctness proof was produced. This brings prospects within reach that systems can be constructed on error-free kernels. That said, it should be kept in mind that as of yet the seL4 ecosystem is in certain important aspects rather limited. For example, although multiprocessor support has been considered a multiprocessor version of seL4 is not available. Moreover, the discussed clustered multi-kernel model raises questions as to the implications on the user-level programming model. In a similar vein, the VMM shipping with seL4 [23] does not support the ARM architecture, which renders it unsuitable for mobile devices. In any event, it will be interesting to examine seL4 and watch its ecosystem evolve.

Uncovering Mobile Rootkits in Raw Memory Our first defense mechanism, comprises of a snapshotting mechanism that allows us to capture both a memory and architectural register state of a VM. To demonstrate the feasibility of our architecture, we designed and built a complementary rootkit detector. In contrast to most of the existing VMI solutions that are based on existing HVs and use their API, our rootkit detector is as an extension to the HV developed at the chair of SecT. This allowed us to extend functionality and expose specific architectural components (e.g. critical system registers and address vectors) of the guest system to our detector VM. As a result, we can inspect additional components apart from the pure guest memory. Several rootkits use exactly these architectural components to hide their presence, and with our scheme we are able to detect them. Additionally, to the best of our knowledge, this is the first research that proposes an architecture specifically designed for ARM devices and covers mobile specific threats.

Owing to ARM's virtualization technology, our architecture provides solid performance and is non-intrusive, allowing to run unmodified guest OSs. The argument that virtualization incurs too much performance degradation on mobile platforms has been proven to be unfounded as various benchmarks showed only a minor performance impact.

In line with our initial statement (see Chapter 1.1) the mechanism provides a very simple interface to expose the memory of the host VM to the detector VM. Moreover, the detector VM is only able to read from the memory snapshot but cannot change anything.

Hypervisor-based Execution Prevention We introduced a mechanism that effectively protects against attack vectors used by kernel rootkit: kernel code injection, kernel code modification, and execution of user code with kernel privileges. Our architecture is based on a slim statically partitioned HV running on an ARM Cortex A7 with hardware VE. The key idea of the execution prevention mechanism is to leverage the VMID feature, originally intended to isolate different VMs from each other, to enforce different execution permissions for PL0 and PL1. The changes necessary for the guest are minimal and only require trivial changes, which allows us to retrofit older OS versions with ease. Our experiments indicated that the incurred overhead, while significant on micro-benchmarks, is below 3% in application benchmarks. With our architecture, we have demonstrated that virtualization does not only offer strong isolation between VMs but can also be leveraged to bolster security within a VM. Also, our work shows that virtualization on mobile devices is feasible and has good chances to proliferate as soon as more devices are equipped with processors featuring ARM VE.

Since porting a guest is uncomplicated, the execution prevention mechanism might even be an alternative to retrofitting security functionality directly into older kernels, some of which might be in use for years after their active development has ceased. As a case in point, researchers demonstrated that Linux' support for Privileged Execute Never (PXN) [6] is insufficient [64], leaving all currently deployed Linux versions up to version 3.18 vulnerable to kernel code execution attacks. With our execution prevention, we were able to prevent that attack scenario even for older kernels that lack PXN support completely, by retrofitting a PXN-like mechanism and limiting the executable pages to only genuine kernel code pages, requiring only minimal changes to the guest.

9 Future Work

The ARM processor architecture steadily evolves, and ARM constantly pushes updates for its processors. In 2011, ARM announced a new processor generation – ARMv8. It provided 64 bit support, consolidated the processor modes, and also updated the hardware VE. The first device with an ARMv8 SoC arrived in 2013 (the iPhone 5s), the first development board with an ARMv8 based SoC followed in 2015. Just recently in 2016, with the announcement of processor generation ARMv8.3, ARM added not only hardware support for nested virtualization but also a pointer authentication security mechanism [88], suggested by Qualcomm. Also, the first processor (Cortex-R52) with real-time capabilities and hardware VE entered the market. This constant stream of innovations and novel mechanisms gives the community (SoC integrators, hard- and software developers, and product designers) new ways and opportunities to bring devices with improved security to the market. Especially, by equipping an increasing number of its processors with hardware VE, ARM created a solid foundation where system designers can build upon to create secure systems without terribly impacting the performance.
But, extending the hardware alone is not enough. To protect the users from attacks, it is important that future OSs leverage these facilities to prevent further growth in the number of machines infected by malware.

When we recap the topics discussed in this thesis and analyze whether the issues we point out in Chapters 4 and 5 have been addressed since we first uncovered them, it becomes clear that the systems are as vulnerable as its predecessors.
The attacks on the virtualization layer when running a Linux OS (discussed in Chapter 4) have already been further explored for the ARMv8 processor architecture in the paper this chapter is based on [22]. We show that all attack vectors (except for one) work on an ARMv8-based system in the same manner as on the ARMv7 architecture. Only the attack vector where Linux is migrated from the secure to the non-secure world is prevented. While consolidating the processor modes, ARM moved the mon mode into a new privilege level (EL3) while; the secure svc mode is still part of EL1. A seamless switch between secure svc mode and mon mode (as was possible on ARMv7) is not possible anymore, thus preventing the attack vector.
As for the covert channels discussed in Chapter 5, further refinements already suggest that we can increase the bandwidth of the covert channels in the future. Also, in this thesis we only discussed covert channels between native L4 processes

built with L4Re. However, in the paper this chapter is based on [87], we show that we can form similar covert channels between L4Linux entities.
Moreover, some of our covert channels scale very well on SMP systems and the trend to faster processors with more cores suggests that the capacity of the described channel types will only grow in the future. At the time of the experiments with these covert channels, most ARM-based devices had a maximum of two CPU cores. Nowadays the, e.g., Cortex-A73 has four CPU cores, and can even be integrated in a big.LITTLE manner, in combination with a Cortex-A53 or Cortex-A35, which leads to ARM-based systems with up to eight cores. Utilizing a device with such a processor system would probably result in very high bandwidth covert channels, giving the conspirators the opportunity to exchange huge amounts of data in a very short time frame.

The above examples illustrate that there is still need for security mechanisms in today's OS. On the other hand, the defense mechanisms discussed throughout this thesis focus on the ARMv7 processor architecture which is already declared as "superseded" by ARM. Its market share will shrink, while an increasing number of devices will feature an ARMv8 processor. In the future for these mechanisms to provide a relevant solution a port to a device with an ARMv8 processor, generation is indispensable.
The challenges of porting the execution prevention (described in Chapter 7) to a device with an ARMv8 processor, concerns adapting the mechanism to the new ARMv8 Stage 2 page table format. Also, currently the mechanism is tailored towards Linux and the virtual address space of a 64 bit Linux is laid out differently than on a 32 bit Linux. Thus the execution prevention would also need to adapt to these new parameters. However, both are only minor implementation changes and in general, do not hinder the execution prevention mechanism to work on an ARMv8 device.
Porting the rootkit detection (described in Chapter 6) would comprise a bigger challenge. The mechanism relies on specific Linux kernel data structures to be identified in raw memory, and it remains to be analyzed in the future on how far these structures differ between a 32 and 64 bit Linux kernel. Consequently, the set of user space tools that work on the raw memory snapshot of a VM would need to be reworked towards the specific quirks of a 64 bit Linux kernel.

Apart from porting the mechanisms to the ARMv8 processor architecture, both prototypes can be evolved into the following directions. The data-only malware proposed by Vogl et al. [111] builds on the ROP [98] mechanism and also achieves persistence while the system is running. Our execution prevention mechanism cannot prevent an adversary from performing a ROP attack in the kernel because the adversaries still executes genuine kernel code. Once the adversary was able to locate a vulnerability in a kernel interface (e.g. driver, kernel subsystem, etc.) that allows him to divert the flow of execution, he can launch a ROP attack. Neither of

our security mechanisms can prevent control flow hijacking attacks; it remains an open issue. In the future, it would be interesting to pair our execution prevention with a control-flow integrity mechanism [121, 29] to rule out ROP attacks. Also, combining our orthogonal rootkit detector with the execution prevention and testing the resulting architecture against real-world rootkits still stands out.

Bibliography

[1] *A Guide to Understanding Covert Channel Analysis of Trusted Systems (Light Pink Book)*. Rainbow Series;NCSC-TG-030. Computer Security Center (CSC), Department of Defense (DoD). Nov. 1993 (cit. on p. 37).

[2] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. „Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References“. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 1248–1259 (cit. on p. 3).

[3] Tiago Alves and Don Felton. „TrustZone: Integrated Hardware and Software Security“. In: *ARM white paper* 3.4 (2004), pp. 18–24 (cit. on p. 14).

[4] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. „A Secure and Reliable Bootstrap Architecture“. In: *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997, pp. 65–71 (cit. on p. 66).

[5] Andrea Arcangeli, Izik Eidus, and Chris Wright. „Increasing memory density by using KSM“. In: *Proceedings of the Linux Symposium*. Citeseer. 2009, pp. 19–28 (cit. on pp. 17, 27).

[6] *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. Whitepaper. ARM Limited, July 2012 (cit. on pp. 11, 12, 14, 26, 28, 66, 81).

[7] *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. Whitepaper. ARM Limited, July 2014 (cit. on pp. 11, 28).

[8] *ARM CoreLink TZC-400 TrustZone Address Space Controller - Technical Reference Manual*. Whitepaper. ARM Limited, July 2013 (cit. on p. 15).

[9] *ARM Dual-Timer Module (SP804)*. Technical Reference Manual. ARM Limited, Jan. 2004 (cit. on pp. 11, 29).

[10] *ARM Generic Interrupt Controller. Architecture version 2.0*. Whitepaper. ARM Limited, July 2013 (cit. on p. 11).

[12] *ARM Security Technology - Building a Secure System using TrustZone Technology*. Whitepaper. ARM Limited, Apr. 2009 (cit. on pp. 11, 14).

[13] Lev Aronsky. *KNOXout - Bypassing Samsung KNOX*. Whitepaper. Viral Security Group, Oct. 2016 (cit. on p. 26).

[14] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. „Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world“. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 90–102 (cit. on p. 20).

[15] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. „The VMware Mobile Virtualization Platform: Is that a Hypervisor in your Pocket?“ In: *ACM SIGOPS Operating Systems Review* 44.4 (2010), pp. 124–135 (cit. on p. 4).

[16] Mick Bauer. „Paranoid penguin: an introduction to Novell AppArmor“. In: *Linux Journal* 2006.148 (2006), p. 13 (cit. on p. 4).

[17] Michael Becher, Maximillian Dornseif, and Christian N Klein. „FireWire: all your memory are belong to us“. In: *Proceedings of CanSecWest* (2005) (cit. on p. 67).

[18] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. „Rootkits on Smart Phones: Attacks, Implications and Opportunities“. In: *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. HotMobile '10. ACM, 2010, pp. 49–54 (cit. on p. 53).

[20] Adam Boileau. „Hit by a bus: Physical access attacks with Firewire“. In: *Presentation, Ruxcon* (2006), p. 3 (cit. on p. 67).

[21] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. „Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector“. In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 987–1004 (cit. on p. 17).

[22] Robert Buhren, Julian Vetter, and Jan Nordholz. „The Threat of Virtualization: Hypervisor-based Rootkits on the ARM Architecture“. In: *18th International Conference on Information and Communications Security (ICICS2016)*. Springer. 2016 (cit. on p. 83).

[24] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. „Towards a systematic study of the covert channel attacks in smartphones“. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2014, pp. 427–435 (cit. on p. 37).

[25] Peter M Chen and Brian D Noble. „When virtual is better than real [operating system relocation to virtual machines]“. In: *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE. 2001, pp. 133–138 (cit. on p. 19).

[27] *Cortex-A7 MPCore*. Technical Reference Manual. ARM Limited, May 2012 (cit. on p. 11).

[28] *Cortex-A9*. Technical Reference Manual. ARM Limited, June 2012 (cit. on p. 11).

[29] John Criswell, Nathan Dautenhahn, and Vikram Adve. „KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels“. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. IEEE Computer Society, 2014, pp. 292–307 (cit. on p. 85).

[35] Christoffer Dall and Jason Nieh. „KVM/ARM: Experiences Building the Linux ARM Hypervisor“. In: (2013) (cit. on p. 25).

[36] Christoffer Dall and Jason Nieh. „KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor“. In: (2014), pp. 333–347 (cit. on p. 25).

[37] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. „Privilege Escalation Attacks on Android“. In: *International Conference on Information Security*. Springer. 2010, pp. 346–360 (cit. on p. 3).

[38] Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. „Cloaker: Hardware supported rootkit concealment“. In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 296–310 (cit. on pp. 23, 54, 58).

[41] *Dhrystone Benchmarking for ARM Cortex Processors*. Whitepaper. ARM Limited, July 2011 (cit. on p. 73).

[42] Kurt Dietrich and Johannes Winter. „Towards Customizable, Application Specific Mobile Trusted Modules“. In: *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*. STC '10. Chicago, Illinois, USA: ACM, 2010, pp. 31–40 (cit. on p. 66).

[43] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. „QUIRE: Lightweight Provenance for Smart Phone Operating Systems.“ In: *USENIX Security Symposium*. Vol. 31. 2011 (cit. on p. 3).

[44] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. „Virtuoso: Narrowing the semantic gap in virtual machine introspection“. In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 297–312 (cit. on p. 20).

[45] Kevin Elphinstone and Gernot Heiser. „From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?“ In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 133–150 (cit. on pp. 4, 80).

[48] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. „AUTOSAR–A Worldwide Standard is on the Road“. In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 62. 2009 (cit. on p. 5).

[49] Tal Garfinkel, Mendel Rosenblum, et al. „A Virtual Machine Introspection Based Architecture for Intrusion Detection.“ In: *NDSS*. Vol. 3. 2003, pp. 191–206 (cit. on pp. 19, 53).

[50] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. „Sprobes: Enforcing kernel code integrity on the trustzone architecture“. In: *arXiv preprint arXiv:1410.7747* (2014) (cit. on p. 20).

[52] Daniel Gruss, David Bidner, and Stefan Mangard. „Practical Memory Deduplication Attacks in Sandboxed JavaScript“. In: *European Symposium on Research in Computer Security*. Springer. 2015, pp. 108–122 (cit. on p. 17).

[53] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. „Cache template attacks: Automating attacks on inclusive last-level caches“. In: *24th USENIX Security*. 2015, pp. 897–912 (cit. on p. 37).

[55] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. „Difference Engine: Harnessing Memory Redundancy in Virtual Machines“. In: *Communications of the ACM* 53.10 (2010), pp. 85–93 (cit. on p. 17).

[56] Brian Hay and Kara Nance. „Forensics Examination of Volatile System Data Using Virtual Introspection“. In: *SIGOPS Oper. Syst. Rev.* 42.3 (Apr. 2008), pp. 74–82 (cit. on pp. 19, 54).

[57] Gernot Heiser and Ben Leslie. „The OKL4 microvisor: convergence point of microkernels and hypervisors“. In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. ACM. 2010, pp. 19–24 (cit. on p. 5).

[58] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005 (cit. on p. 54).

[59] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. „Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones“. In: *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*. IEEE. 2008, pp. 257–261 (cit. on p. 4).

[62] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. „Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction“. In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 128–138 (cit. on pp. 19, 53).

[63] *Kaspersky Security Bulletin 2014 – A Look into the APT Crystal Ball*. Whitepaper. Kaspersky Lab, Dec. 2014 (cit. on p. 3).

[64] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. „ret2dir: Rethinking Kernel Isolation“. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 957–972 (cit. on pp. 18, 66, 81).

[65] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. „kGuard: Lightweight Kernel Protection against Return-to-User Attacks“. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 459–474 (cit. on pp. 18, 66).

[66] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. „Comprehensive Formal Verification of an OS Microkernel“. In: *ACM Trans. Comput. Syst.* 32.1 (Feb. 2014), 2:1–2:70 (cit. on p. 80).

[67] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. „seL4: Formal Verification of an OS Kernel“. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220 (cit. on p. 80).

[68] Tobias Klein. *Rootkit Profiler LX*. Tech. rep. www.trapkit.de, Apr. 2007 (cit. on pp. 24, 53).

[71] Butler W. Lampson. „A note on the confinement problem“. In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615 (cit. on p. 37).

[72] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. „L4Android: a generic operating system framework for secure smartphones“. In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pp. 39–50 (cit. on p. 5).

[73] Jochen Liedtke. *On micro-kernel construction*. Vol. 29. 5. ACM, 1995 (cit. on p. 5).

[74] Yuqi Lin, Liping Ding, Jingzheng Wu, Yalong Xie, and Yongji Wang. „Robust and Efficient Covert Channel Communications in Operating Systems: Design, Implementation and Evaluation“. In: *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE. 2013, pp. 45–52 (cit. on p. 37).

[75] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. „Andrubis–1,000,000 Apps later: A view on current Android malware behaviors“. In: *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*. IEEE. 2014, pp. 3–17 (cit. on p. 3).

[76] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. „Xtratum: a hypervisor for safety critical embedded systems“. In: *11th Real-Time Linux Workshop*. Citeseer. 2009, pp. 263–272 (cit. on p. 4).

[77] Paul E. Mckenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. „Read-Copy Update“. In: *In Ottawa Linux Symposium*. 2001, pp. 338–367 (cit. on p. 47).

[78] Larry W McVoy, Carl Staelin, et al. „lmbench: Portable Tools for Performance Analysis.“ In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294 (cit. on p. 34).

[79] Roberto Mijat and Andy Nightingale. „Virtualization is coming to a platform near you“. In: *ARM White Paper* (2011) (cit. on pp. 6, 29).

[80] Jonathan K Millen. „Covert Channel Capacity.“ In: *IEEE Symposium on Security and Privacy*. 1987 (cit. on p. 37).

[83] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. „seL4: from general purpose to a proof of information flow enforcement“. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 415–429 (cit. on p. 4).

[84] Jon Oberheide and Charlie Miller. „Dissecting the android bouncer“. In: *SummerCon2012, New York* (2012) (cit. on p. 3).

[85] Keisuke Okamura and Yoshihiro Oyama. „Load-based Covert Channels Between Xen Virtual Machines“. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: ACM, 2010, pp. 173–180 (cit. on p. 37).

[87] Michael Peter, Matthias Petschick, Julian Vetter, Jan Nordholz, Janis Danisevskis, and J-P Seifert. „Undermining Isolation through Covert Channels in the Fiasco.OC Microkernel“. In: *Information Sciences and Systems 2015*. Springer, 2016, pp. 147–156 (cit. on p. 84).

[88] *Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions*. Whitepaper. Qualcomm Technologies, Inc., Jan. 2017 (cit. on p. 83).

[89] Paul J Prisaznuk. „ARINC 653 role in integrated modular avionics (IMA)“. In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE. 2008, 1–E (cit. on p. 5).

[90] Ryan Riley, Xuxian Jiang, and Dongyan Xu. „Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing“. In: *Recent Advances in Intrusion Detection*. Springer. 2008, pp. 1–20 (cit. on pp. 20, 65).

[91] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. „Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds“. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 199–212 (cit. on p. 37).

[92] Dan Rosenberg. „QSEE TrustZone kernel integer over flow vulnerability“. In: *Black Hat conference*. 2014 (cit. on p. 26).

[93] John M Rushby. *Design and verification of secure systems*. Vol. 15. 5. ACM, 1981 (cit. on p. 5).

[94] Joanna Rutkowska. „Introducing blue pill“. In: *The official blog of the invisiblethings. org* 22 (2006) (cit. on p. 23).

[96] *Secure Virtual Machine Architecture Reference Manual*. Whitepaper. AMD, May 2005 (cit. on p. 4).

[97] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. „SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes“. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 335–350 (cit. on pp. 4, 20, 65).

[98] Hovav Shacham. „The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)“. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561 (cit. on pp. 67, 84).

[99] Di Shen. „Exploiting Trustzone on Android“. In: *Black Hat conference*. 2015 (cit. on p. 26).

[102] Patrick Stewin and Iurii Bystrov. „Understanding DMA malware“. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 21–41 (cit. on p. 67).

[103] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. „Memory Deduplication As a Threat to the Guest OS“. In: *Proceedings of the Fourth European Workshop on System Security*. EUROSEC '11. Salzburg, Austria: ACM, 2011, 1:1–1:6 (cit. on pp. 17, 37).

[104] *TCG Mobile Trusted Module Specification*. White Paper. Specification Version 1.0. Trusted Computing Group, June 2008 (cit. on p. 66).

[108] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung, and Larry Smith. „Intel virtualization technology“. In: *Computer* 38.5 (2005), pp. 48–56 (cit. on p. 4).

[110] Timothy Vidas, Daniel Votipka, and Nicolas Christin. „All Your Droid Are Belong to Us: A Survey of Current Android Attacks“. In: *WOOT*. 2011, pp. 81–90 (cit. on p. 3).

[111] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. „Persistent data-only malware: Function Hooks without Code“. In: *Symposium on Network and Distributed System Security (NDSS)*. 2014 (cit. on p. 84).

[112] Carl A Waldspurger. „Memory Resource Management in VMware ESX Server“. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 181–194 (cit. on p. 17).

[113] Zhi Wang and Xuxian Jiang. „Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity“. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 380–395 (cit. on p. 67).

[114] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. „Linux Security Modules: General Security Support for the Linux Kernel.“ In: *USENIX Security Symposium*. Vol. 2. 2002, pp. 1–14 (cit. on p. 4).

[115] Jingzheng Wu, Liping Ding, Yuqi Lin, Nasro Min-Allah, and Yongji Wang. „Xenpump: a new method to mitigate timing channel in cloud computing“. In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE. 2012, pp. 678–685 (cit. on p. 79).

[116] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. „Security implications of memory deduplication in a virtualized environment“. In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE. 2013, pp. 1–12 (cit. on pp. 17, 37).

[117] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. „An Exploration of L2 Cache Covert Channels in Virtualized Environments“. In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW '11. Chicago, Illinois, USA: ACM, 2011, pp. 29–40 (cit. on p. 37).

[118] 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, pp. 569–584 (cit. on p. 20).

[121] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. „Practical Control Flow Integrity and Randomization for Binary Executables“. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. IEEE Computer Society, 2013, pp. 559–573 (cit. on p. 85).

[122] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. „CacheKit: Evading Memory Introspection Using Cache Incoherence“. In: *European Symposium on Security and Privacy, 2016, IEEE*. IEEE. 2016 (cit. on pp. 23, 54).

Online

[11] ARM Ltd. *mbed TLS*. Accessed: 2015-05-26. Jan. 2013. URL: `https://tls.mbed.org/` (cit. on p. 61).

[19] Michael Boelen. *Rootkit Hunter*. Accessed: 2017-02-15. Apr. 2015. URL: `http://rkhunter.sourceforge.net/` (cit. on pp. 24, 53).

[23] *CAmkES*. Accessed: 2017-02-15. July 2014. URL: `https://wiki.sel4.systems/CAmkES` (cit. on p. 80).

[26] Michael Coppola. *Suterusu Rootkit: Inline Kernel Function Hooking on x86 and ARM*. Accessed: 2016-03-08. Jan. 2013. URL: `http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/` (cit. on p. 23).

[30] *Cubieboard 2*. Accessed: 2015-05-06. URL: http://cubieboard.org/model/cb2/ (cit. on pp. 12, 34).

[31] *Cubietruck*. Accessed: 2015-05-18. URL: http://cubieboard.org/model/cb3/ (cit. on pp. 12, 72).

[32] CVE Details: The ultimate security vulnerabilty datasource. *Linux Kernel: Vulnerability Statistics*. Accessed: 2016-03-29. Mar. 2016. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 (cit. on pp. 19, 24, 53).

[33] CVE Details: The ultimate security vulnerabilty datasource. *Microsoft Windows : Security Vulnerabilities*. Accessed: 2017-02-10. Feb. 2017. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-3435/Microsoft-Windows.html (cit. on p. 19).

[34] CVE Details: The ultimate security vulnerabilty datasource. *XEN : Security Vulnerabilities*. Accessed: 2017-02-10. Feb. 2017. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html (cit. on p. 19).

[39] denx - software engineering. *Das U-Boot – the Universal Boot Loader*. Accessed: 2016-04-29. Apr. 2016. URL: http://www.denx.de/wiki/U-Boot (cit. on p. 26).

[40] Hitesh Dharmdasani. *Android-Rootkit*. Accessed: 2015-04-13. 2015. URL: https://github.com/hiteshd/Android-Rootkit (cit. on pp. 23, 60).

[46] *Fiasco.OC website*. Accessed: 2016-09-05. June 2016. URL: http://os.inf.tu-dresden.de/fiasco/ (cit. on pp. 5, 39, 47).

[47] Michael Flossman. *ViperRAT: The mobile APT targeting the Israeli Defense Force that should be on your radar*. Accessed: 2017-04-03. Lookout, Inc. Feb. 2017 (cit. on p. 3).

[51] Dan Goodin. *Found: Quite possibly the most sophisticated Android espionage app ever*. Accessed: 2017-04-10. Apr. 2017. URL: https://arstechnica.com/security/2017/04/found-quite-possibly-the-most-sophisticated-android-espionage-app-ever/ (cit. on p. 3).

[54] Sebastian Guerrero. *Getting sys_call_table on Android*. Mar. 2013. URL: https://www.nowsecure.com/blog/2013/03/13/syscalltable-android-playing-rootkits/ (cit. on p. 57).

[60] *In-kernel memory compression*. Accessed: 2016-05-26. URL: https://lwn.net/Articles/545244/ (cit. on p. 28).

[61] *iOS Security (iOS 9.3 or later)*. Accessed: 2016-10-11. May 2016. URL: https://www.apple.com/business/docs/iOS_Security_Guide.pdf (cit. on p. 5).

[69] *L4Linux - Running Linux on top of L4*. Accessed: 2017-02-15. May 2014. URL: http://l4linux.org (cit. on pp. 5, 80).

[70] *L4Re Runtime Environment*. Accessed: 2017-02-15. May 2014. URL: http://os.inf.tu-dresden.de/L4Re (cit. on pp. 39, 47, 80).

[81] mncoppola. *An LKM rootkit targeting Linux 2.6/3.x on x86(_64), and ARM*. Accessed: 2015-04-13. Sept. 2014. URL: https://github.com/mncoppola/suterusu (cit. on pp. 54, 55, 59, 60).

[82] Nelson Murilo and Klaus Steding-Jessen. *chkrootkit - locally checks for signs of a rootkit*. Accessed: 2017-02-15. Apr. 2015. URL: http://www.chkrootkit.org/ (cit. on pp. 24, 53).

[86] *PandaBboard Technical Specs*. Accessed: 2017-02-15. May 2014. URL: http://pandaboard.org/content/platform (cit. on p. 12).

[95] sd and devik. *Linux on-the-fly kernel patching without LKM*. Accessed: 2017-02-15. Dec. 2001. URL: http://phrack.org/issues/58/7.html (cit. on pp. 54, 55).

[100] Steven Sinofsky. *Reducing runtime memory in Windows 8*. Accessed: 2017-02-14. Oct. 2011. URL: https://blogs.msdn.microsoft.com/b8/2011/10/07/reducing-runtime-memory-in-windows-8/ (cit. on p. 17).

[101] Lennart Sorensen. *TI SMC call*. Accessed: 2016-05-12. Texas Instruments. 2015. URL: https://git.ti.com/ti-linux-kernel/ti-linux-kernel/blobs/master/arch/arm/mach-omap2/omap-headsmp.S\#line60 (cit. on p. 26).

[105] Trend Micro Inc. *OSSEC*. Accessed: 2017-02-15. Apr. 2015. URL: http://www.ossec.net/?page_id=19 (cit. on p. 53).

[106] trimpsyw. *adore-ng - linux rootkit adapted for 2.6 and 3.x*. Accessed: 2015-04-13. Oct. 2014. URL: https://github.com/trimpsyw/adore-ng (cit. on pp. 23, 59).

[107] truff. *Infecting loadable kernel modules*. Accessed: 2017-02-15. Aug. 2003. URL: http://phrack.org/issues/61/10.html (cit. on pp. 54, 55).

[109] unixfreaxjp. *MMD-0028-2014 - Fuzzy reversing a new China ELF "Linux/XOR.DDoS"*. Accessed: 2016-03-08. Sept. 2014. URL: http://blog.malwaremustdie.org/2014/09/mmd-0028-2014-fuzzy-reversing-new-china.html (cit. on pp. 23, 55).

[119] dong-hoon you. *Android platform based linux kernel rootkit*. Accessed: 2017-02-15. Apr. 2011. URL: http://phrack.org/issues/68/6.html (cit. on pp. 54, 55, 57, 59).

[120] Zeppoo. *Zeppoo - Anti Rootkit Software*. Accessed: 2015-05-21. May 2013. URL: http://sourceforge.net/projects/zeppoo/ (cit. on p. 53).

[123] Yanmin Zhang. *Hackbench*. Accessed: 2016-09-06. 2008. URL: https://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c (cit. on p. 34).

Acronyms

ARM	**A**dvanced **R**isk **M**achines
ASID	**A**ddress **S**pace **ID**entifier
CPSR	**C**urrent **P**rocessor **S**tatus **R**egister
CPU	**C**entral **P**rocessing **U**nit
FPU	**F**loating **P**oint **U**nit
GIC	**G**eneric **I**nterrupt **C**ontroller
HV	**H**yper **V**isor
IDS	**I**nstrusion **D**etection **S**ystem
IPA	**I**ntermediate **P**hysical **A**ddress
IPS	**I**nstrusion **P**revention **S**ystem
KIP	**K**ernel **I**nfo **P**age
KSM	**K**ernel **S**ame-Page **M**erging
KVM	**K**ernel-based **V**irtual **M**achine
LKM	**L**oadable **K**ernel **M**odule
LLC	**L**ast **L**evel **C**ache
MMU	**M**emory **M**anagement **U**nit
OS	**O**perating **S**ystem
PC	**P**rogramm **C**ounter
PID	**P**rocess **ID**entifier
PL	**P**rivilege **L**evel
PT	**P**age **T**ables
RCU	**R**ead **C**opy Update
SGX	**S**oftware **G**uard **E**xtensions
SMP	**S**ymmetric **M**ulti **P**rocessing
SOC	**S**ystem **O**n **C**hip
TCB	**T**rusted **C**omputing **B**ase
TLB	**T**ranslation **L**ookaside **B**uffer
TPM	**T**rusted **P**latform **M**odule
TZ	**T**rust **Z**one
UP	**U**ni **P**rocessing
UTCB	**U**ser-level **T**ask **C**ontrol **B**lock
VM	**V**irtual **M**achine
VMI	**V**irtual **M**achine **I**ntrospection
VMID	**V**irtual **M**achine **ID**entifier
VMM	**V**irtual **M**achine **M**onitor
VCPU	**V**irtual **C**PU
VE	**V**irtualization **E**xtensions

List of Figures

List of Tables