JYU DISSERTATIONS 406

Raz Ben Yehuda

Manipulating the ARM Hypervisor and TrustZone

UNIVERSITY OF JYVÄSKYLÄ
FACULTY OF INFORMATION TECHNOLOGY

JYU DISSERTATIONS 406

Raz Ben-Yehuda

Manipulating the ARM Hypervisor and TrustZone

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi elokuun 6. päivänä 2021 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
on August 6, 2021 at 12 o'clock noon.

JYVÄSKYLÄN YLIOPISTO
UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2021

Editors
Timo Männikkö
Faculty of Information Technology, University of Jyväskylä
Timo Hautala
Open Science Centre, University of Jyväskylä

ISBN 978-951-39-8752-7 (PDF)
URN:ISBN:978-951-39-8752-7
ISSN 2489-9003

Permanent link to the online version of this publication: http://urn.fi/URN:ISBN:978-951-39-8752-7

ABSTRACT

Ben Yehuda, Raz
Manipulating the ARM Hypervisor and TrustZone
Jyväskylä: University of Jyväskylä, 2021, 76 p.
(JYU Dissertations
ISSN 2489-9003; 406)
ISBN 978-951-39-8752-7 (PDF)

ARM architecture keeps extending, and new features are added in each edition of this processor's architecture. We examine the various techniques to manipulate the ARM hypervisor. In this work, we present a new execution context in the Linux operating system, which we refer to as the hyplet. The hyplet is a technique in which a function of a regular Linux process is executed in the hypervisor. It is through the use of the hyplet that an additional security layer is put inside an executing Linux process, inaccessible to common user space or kernel space privileges. Also, the hyplet provides an infrastructure for a CFI (Control Flow Inspection) technique named C-FLAT, a virtual disk used to trap intruders (honeypot), and a method to acquire coherent memory images for forensics. The acquisition is performed slowly, thereby reduces heat and power, and therefore a good solution for battery-based devices such as smartphones. Also, we show that the hyplet, compared to other RPC (Remote Procedure Call) techniques, provides an extremely fast RPC among Linux Processes. Through the hyplet, it is also possible to execute ISR (interrupt service routine) in a regular user-space Linux process. In Linux it is possible to offload a processor, usually to reduce power. We combined offloading a processor and the hyplet to demonstrate hard real-time. This technology is referred to as the offline hyplet. The offline hyplet demonstrates high-resolution timers, 20Khz, on a relatively slow ARM processor, executing a userspace routine inside a regular Linux process. Other than that, our research presents the hyperwall, a technology to protect network cards. Lastly, we provide a tutorial for a DMA attack on TrustZone running the OP-TEE operating system.

Keywords: Hypervisor, TrustZone, ARM, Virtualization, Real-time, Safety

TIIVISTELMÄ (ABSTRACT IN FINNISH)

Ben Yehuda, Raz
ARM-hypervisorin ja TrustZonen käsittely
Jyväskylä: University of Jyväskylä, 2021, 76 s.
(JYU Dissertations
ISSN 2489-9003; 406)
ISBN 978-951-39-8752-7 (PDF)

ARM-Architecture jatkuu jatkuvasti, ja uusia ominaisuuksia lisätään prosessorin jokaisessa versiossa. Tutkimme erilaisia tekniikoita ARM-hypervisor manipuloimiseksi. Tässä työssä esitämme uuden suoritusyhteyden Linux-käyttöjärjestelmässä, jota kutsumme Hyplet. Hyplet on tekniikka, jossa tavallisen Linux-prosessin toiminto suoritetaan hypervisor. Hyplet avulla laitetaan suoritettavan Linux-prosessin sisään ylimääräinen turvakerros, johon ei pääse tavalliselle käyttäjä- tai ydintilaa koskeville oikeuksille. Hyplet tarjoaa myös infrastruktuurin CFI (Control Flow Inspection) -tekniikalle, nimeltään C-FLAT, virtuaalilevylle, jota käytetään tunkeilijoiden ansaan (hunajapotti), ja menetelmän yhtenäisten muistikuvien hankkimiseksi Forensics. Hankinta suoritetaan hitaasti, mikä vähentää lämpöä ja virtaa ja on siten hyvä ratkaisu akkupohjaisille laitteille, kuten älypuhelimille. Näytämme myös, että hypletti tarjoaa muihin RPC (Remote Procedure Call) -tekniikoihin verrattuna erittäin nopean RPC: n Linux-prosessien joukossa. Hyplet kautta on myös mahdollista suorittaa ISR (keskeytä palvelurutiini) tavallisessa user-space Linux -prosessissa. Linuxissa on mahdollista purkaa prosessori, yleensä virran vähentämiseksi. Yhdistimme prosessorin ja hyplet purkamisen osoittaaksemme kovaa reaaliaikaista. Tätä tekniikkaa kutsutaan offline-hyplet. Offline-hypletti osoittaa korkean resoluution ajastimia, 20khz, suhteellisen hitaalla ARM-prosessorilla, joka suorittaa käyttäjätilan rutiinia tavallisessa Linux-prosessissa. Tämän lisäksi tutkimuksemme esittelee Hyperwall-tekniikkaa, joka suojaa verkkokortteja. Viimeiseksi tarjoamme opetusohjelman DMA-hyökkäykselle TrustZoneen, joka käyttää OP-TEE-käyttöjärjestelmää.

Avainsanat: Hypervisor, TrustZone, ARM, virtualisointi, reaaliaika, turvallisuus

Author

Raz Ben Yehuda
University of Jyväskylä
Finland

Supervisors

Dr. Nezer Jacob Zaidenberg
University of Jyväskylä
Finland

College of Management Academic Studies
Israel

Professor Timo Hämäläinen
University of Jyväskylä
Finland

Professor Pekka Neittaanmaki̇
University of Jyväskylä
Finland

Reviewers

Miguel Pupo Correia
Universidade de Lisboa
Portugal

Professor Aurélien Francillon
Eurecom
France

Opponent

Professor Christian Grothoff
Bern University of Applied Sciences
Switzerland

ACKNOWLEDGEMENTS

I am grateful to Dr. Nezer Zaidenberg for giving me this opportunity and guiding me through this journey. I'd also like to thanks Professor Timo Hämäläinen and Professor Pekka Neittaanmaki for opening the academy gates to me.

ACRONYMS

TEE	Trusted Execution Environment
RICH-OS	Rich operating system
HYP	Hypervisor
REE	Rich Execution Environment
ELx	Exception Level x
DMA	Direct Memory Access
TZPC	TrustZone Protection Controller
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
SMC	System Monitor Call
HVC	Hypervisor Call
SVC	Supervisor Call
ACE	Arbitrary Code Execution
CFI	Control Flow Inspection
ASLR	Address Space Layout Randomization
TA	Trusted Application
SLAT	Second Level Address Translation
IPA	Intermediate Physical Address
EPT	Extended Page Table
RISC	Reduced Instruction Set
CISC	Complex Instruction Set
MMU	Memory Management Unit
GPOS	Generic Operating System
ERET	Exception Return
NS	Non secure bit
SP	Secure Physical
SGX	Software Guard Extensions
PMU	Performance management unit
ROP	Return Oriented Programming
PAC	Pointer Authentication Code
IPC	Inter-process communication
GPA	Guest Physical Address
QSEE	Qualcomm Secure Execution Environment
RTOS	Real Time Operating System
KVM	Kernel Virtual Machine
LLC	Last Level Cache
SMMU	System Memory Management Unit
RPC	Remote Procedure Call
VGIC	Virtual Interrupt Controller
hypRPC	Hyplet RPC

hypISR	Hyplet ISR
SoM	System on a Module
CVE	Common Vulnerability Error
BKPT	Break Point
NOP	No Operation
OP-TEE	Open Portable Trusted Execution Environment
DEP	Data Execution Prevention

LIST OF FIGURES

FIGURE 1 ARM incline ©Wikipedia 22
FIGURE 2 ARMv8 Architecture ©ARM 23
FIGURE 3 ARMv8 TrustZone Interface 24
FIGURE 4 ARMv8 MMU ©ARM 25
FIGURE 5 ARMv8 Simplified TrustZone Controller ©ARM 26
FIGURE 6 The Incoherent Image Problem 28
FIGURE 7 Raspberry PI3 timer latencies, 1 ms interval 35
FIGURE 8 Hypervisor Types 38
FIGURE 9 Hypervisor Exception Levels 39
FIGURE 10 SLAT in ARM ©Pratt 40
FIGURE 11 ARMv8 Hypervisor Interface 41
FIGURE 12 Translation table crosses Exception Levels 42
FIGURE 13 Address space prior to VHE ©ARM documentation arm (2021a) 42
FIGURE 14 Address space with VHE ©ARM documentation arm (2021a) .. 43
FIGURE 15 func mapped to EL2 and EL0 43
FIGURE 16 Embedded Partitioning 46
FIGURE 17 RPC durations 52
FIGURE 18 1Khz jitter (us) in a Raspberry Pi3 56
FIGURE 19 The hyplet-offlet in various frequencies 56
FIGURE 20 C-FLAT ©abera 58

LIST OF TABLES

TABLE 1 Good Input (ms) for advpng 59
TABLE 2 Erroneous Input (ms) for advpng 60
TABLE 3 Hyperwall LMbench 62
TABLE 4 SLAT Real world load GB/s..... 64
TABLE 5 Duration of stack access in ticks 244

LIST OF INCLUDED ARTICLES

PI Raz Ben Yehuda, Yair Wiseman. The offline scheduler for embedded transportation systems.. *Proceedings of Industrial Technology (ICIT), IEEE International Conference on. 2011*.

PII Raz Ben Yehuda and Nezer Zaidenberg. Hyplets - Multi Exception Level Kernel towards Linux RTOS. *Proceedings of the 11th ACM International Systemsand Storage Conference. 2018*, 2018.

PIII Raz Ben Yehuda, Roee Leon and Nezer Zaidenberg.. ARM security alternatives.. *Proceedings of the European conference on information arfare and security.Academic Conferences International. 2019*, 2019.

PIV Michael Kiperberg , Raz Ben Yehuda and Nezer Zaidenberg. HyperWall: A Hypervisor for Detection and Prevention of Malicious Communication. *International Conference on Network and System Security. Best paper award*, 2020.

PV Raz Ben Yehuda, Nezer Zaidenberg. Protection against reverse engineering in ARM. *Proceedings of Industrial Technology (ICIT), IEEE International Conference on. 2011*, 2011.

PVI Raz Ben Yehuda, Nezer Zaidenberg. The hyplet-Joining a Program and a Nanovisor for real-time and Performance. *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). IEEE*, 2020.

PVII Nezer Zaidenberg, Michael Kiperberg, Raz Ben Yehuda, Roee Leon, Asaf Alagawi and Amit Resh.. Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot.. *Information Systems Security and Privacy. ICISSP 2019. Communications in Computer and Information Science, vol 1221*, 2020.

PVIII Raz Ben Yehuda, Erez Shlingbaum, Shaked Tayouri, Yuval Gershfeld and Nezer Zaidenberg. Hypervisor Memory acquisition for ARM. *Forensic Science International: Digital Investigation. 301106*, 2021.

PIX Raz Ben Yehuda and Nezer Zaidenberg. Offline nanovisor. *submitted*.

PX Raz Ben Yehuda, Adam Aronov, Or Ekstein, Michael Kiperberg and Nezer Zaidenberg. C FLAT nanovised. *submitted*.

PXI Ron Stajnord, Raz Ben Yehuda and Nezer Zaidenberg.. Attacking TrustZone. *Submitted*.

Author contribution

The idea of encrypted code execution in a hypervisor, which is described in articles [PV] (Ben Yehuda and Zaidenberg (2019)) and [PVI] (Ben Yehuda and Zaidenberg (2020)) was devised together with Dr. Nezer Zaidenberg. Work on article [PV] led to [PII] (Ben Yehuda and Zaidenberg (2018)) and [PVI]. In [PV] The author ported TrulyProtect (Resh et al. (2017), Averbuch et al. (2013)) from Intel to the ARM architecture. For this purpose the author designed the Hyplet computation model for ARM.
[PII] and [PVI] describes the hyplet and additional benefits affiliated with the hyplet. The author's contribution to this research are in designing the hyplet, formalizing the idea of the hyplet and its possible uses. The author designed and implemented the decryption system in the hypervisor and describes the design and performance analysis in articles [PV] and [PVI].
Article [PI] (Ben-Yehuda and Wiseman (2011) was devised together was Dr. Yair Weisman and implemented by the author as a means to achieve real-time in regular Debian distribution. An extended version of [PI] is Ben-Yehuda and Wiseman (2013). Papers [PI] and [PVI] led to paper [PIX], which again, was devised together with Dr. Nezer Zaidenberg, and was designed and implemented by the author.
C-FLAT, article [PX], is in the area of CFI, devised together with Dr. Nezer Zaidenberg, and is considered another application of the hyplet. The author's contribution is in creating the infrastructure for the C-FLAT implementation for the other co-authors. Article [PVII] (Zaidenberg et al. (2019)) is an extension of Kiperberg et al. (2019a) . The paper demonstrates another implementation of the hyplet, was devised by Dr. Nezer Zaidenberg, and was created by the author with the collaboration of the other co-authors. The innovation in this paper is adding USB Honeypots (Ben Yehuda et al. (2019a)) to the memory acquisition system.
Article [PVIII] was devised together with Dr. Nezer Zaidenberg, and designed by the author. The author also implemented the infrastructure for the memory acquisition, and on top of it, the co-authors completed the acquisition process implementation and testing.
[PIII] (Ben Yehuda et al. (2019b)) is a survey authored by the author, Dr. Nezer Zaidenberg and Dr. Roee Leon. We present information gather jointly about virtualization as a means to secure ARM-based devices. The paper was also released in an extended version in the Encyclopedia of Cybercrime (Zaidenberg et al. (2020)).
In the article [PIV] (Kiperberg et al. (2020)), Dr. Michael Kiperberg presents the hyperwall as a technology to protect network cards, with the help of Dr. Nezer Zaidenberg. The author's contribution is showing how the hyperwall is implemented on ARM.
Article [PXI] is a joint effort of the author and the co-author, with the supervision of Dr. Nezer Zaidenberg, to show that a SOC that uses TrustZone but doesn't fully comply with ARM TrustZone specifications is penetrable.

CONTENTS

ABSTRACT
TIIVISTELMÄ (ABSTRACT IN FINNISH)
ACKNOWLEDGEMENTS
ACRONYMS
LIST OF FIGURES
LIST OF TABLES
LIST OF INCLUDED ARTICLES
CONTENTS

1 INTRODUCTION 19

2 ARM 21
2.1 Background 21
2.2 Operating Systems in ARM 22
2.3 Security in ARM 23
2.3.1 TrustZone 23
2.3.2 Cybersecurity challenges 26
2.3.2.1 Malware challenges 26
2.3.2.2 Computer Forensics Challenges 27
2.3.2.3 Trusted Execution challenges 28
2.3.2.4 CFI challenges 29
2.3.3 ARM Security Alternatives 30
2.3.3.1 GlobalPlatform 30
2.3.3.2 General Dynamics OKL4 30
2.3.3.3 seL4 microkernel 30
2.3.3.4 Google Trusty TEE 31
2.3.3.5 Linaro OP-TEE 32
2.3.3.6 Kinibi 32
2.3.3.7 Xen 33
2.3.3.8 Xvisor 33
2.3.3.9 QSEE 34
2.4 Real-time 34
2.4.1 Real-time Operating Systems in ARM 34
2.4.2 Real-time challenges 35

3 HYPERVISORS 37
3.1 Overview 37
3.2 ARM Virtualization 38
3.3 Security of thin hypervisors 43
3.3.1 Translation tables protection 44
3.3.2 Cache and TLB 44
3.3.3 Secure interrupts 45
3.3.4 DMA attack 45

3.3.5 CFI attacks 45
3.3.6 Forensics 45
3.4 Real-time Hypervisors 46

4 SUMMARY OF ORIGINAL PAPERS 47
4.1 Reverse Engineering Protection in ARM 47
4.1.1 Paper Details 47
4.1.2 Research Question 47
4.1.3 Technique 48
4.1.4 Results 49
4.1.5 Future work 49
4.2 The hyplet 50
4.2.1 Paper Details 50
4.2.2 Research Question 50
4.2.3 Technique 50
4.2.4 Results 51
4.2.5 Future work 52
4.3 Memory Acquisition 52
4.3.1 Paper Details 52
4.3.2 Research Question 53
4.3.3 Technique 53
4.3.4 Results 54
4.3.5 Future work 54
4.4 The Offline Nanovisor 54
4.4.1 Paper Details 54
4.4.2 Research Question 54
4.4.3 Technology 55
4.4.4 Results 56
4.4.5 Future work 57
4.5 C-FLAT Linux 57
4.5.1 Paper Details 57
4.5.2 Research Question 57
4.5.3 Technique 58
4.5.4 Results 59
4.5.5 Future Work 60
4.6 HyperWall 60
4.6.1 Paper Details 60
4.6.2 Research Question 60
4.6.3 Technique 61
4.6.4 Results 61
4.6.5 Future work 62
4.7 Hypervisor Memory Introspection and Hypervisor based Malware Honeypot 62
4.7.1 Paper Details 63
4.7.2 Research Question 63

4.7.3 Technique 63
4.7.4 Results 63
4.7.5 Future work 64
4.8 Attacking TrustZone 64
4.8.1 Paper Details 64
4.8.2 Research Question 64
4.8.3 Technique 65
4.8.4 Results 66
4.8.5 Future Work 66

5 CONCLUSIONS 67

YHTEENVETO (SUMMARY IN FINNISH) 68

REFERENCES 69

INCLUDED ARTICLES

6 ERRATA 239
6.1 Essential Corrections in the Dissertations 239
6.1.1 C1 239
6.1.2 C2 239
6.1.3 C3 239
6.1.4 C4 240
6.1.5 C5 240
6.1.6 C6 240
6.1.7 C7 240
6.1.8 C8 240
6.1.9 C9 240
6.1.10 C10 240
6.1.11 C11 240
6.1.12 C12 240
6.1.13 C13 240
6.1.14 C14 241
6.1.15 C15 241
6.2 Paper PI: The offline scheduler for embedded transportation systems 241
6.2.1 C16 241
6.2.2 C17 241
6.2.3 C18 241
6.2.4 C19 241
6.2.5 C20 241
6.3 Paper PII: Hyplets - Multi Exception Level Kernel towards Linux RTOS - Systor 242
6.3.1 C21 242
6.3.2 C22 242

6.4 Paper PIII: ARM Security Alternatives 242
6.4.1 C23 242
6.5 Paper PIV: Hyperwall 242
6.5.1 C24 242
6.5.2 C25 242
6.5.3 C26 243
6.5.4 C27 243
6.5.5 C28 243
6.6 Paper PV: Protection against reverse engineering in ARM 243
6.6.1 C29 243
6.6.2 C30 243
6.6.3 C31 243
6.6.4 C32 244
6.6.5 C33 244
6.7 Paper PVI: The hyplet-Joining a Program and a Nanovisor for real-time and Performance - SPECTS 2020, IEEE 244
6.7.1 C34 244
6.7.2 C35 244
6.7.3 C36 245
6.7.4 C37 245
6.7.5 C38 245
6.7.6 C39 245
6.8 Paper PVIII: Hypervisor Memory acquisition for ARM 245
6.8.1 C40 245

1 INTRODUCTION

This dissertation research questions in what ways is it possible to manipulate or break some of the ARM processor extended features:

- Can we manipulate ARM's Virtualization to gain security ?
- Can we manipulate ARM's Virtualization to gain real-time ?
- Is it possible to break Trustzone ?

While gaining security through virtualization in x86 has been researched Averbuch et al. (2013), we looked at ARM-v8 and extended its security arsenal by adding a layer of security in the hypervisor. The same applies for real-time. Real-time via hypervisor had already been shown in ARM and x86 Heiser and Leslie (2010). In this work, we employed different techniques to achieve real-time. Lastly, we provide a tutorial for breaking Arm's Trustzone by a DMA attack.
Linux is a common and widely used operating system and therefore is a target to malicious attackers, so protecting it, setting up traps (honeypots) is a good idea. The abundance of software in Linux also provides a wide attack surface, and thus we looked for additional ways to protect the Linux (and Android) operating systems.
The thesis presents a new execution context called the hyplet. The hyplet is a hybrid between a Nanovisor and an ELF (Executable Linkable Format) userspace program. The hyplet was created first as a means to provide a safe execution environment for part of a Linux ELF program. However, throughout our research, we learned that the hyplet has other uses, such as real-time and in the area of CFI (Control Flow Inspection). The hyplet is designed to extend the operating system offering while changing it as little as possible. As the hyplet is a user space code, and has access to user space data, the user space program can help debug the Nanovisor before it is encrypted.
Our Nanovisor does change the operating system behavior nor its performance. Also, the Nanovisor does not lock the user to our technology, and he or she may choose to neglect the Nanovisor without endangering the data or the code. Our

Nanovisor is a plugin that may be removed any time.
The Nanovisor is an extension or improvement to current technologies. For example, in the memory acquisition paper, we extended the LiME kernel module to acquire the entire RAM more gracefully.
The technologies presented in this thesis do not contradict other available technologies for protection or real-time. For example, the Offline Nanovisor is offered as an RTOS library for Linux in addition to RT PREEMPT, and Hyperwall does not require changes to the Linux kernel.
Since our efforts focus on improvements by manipulating the ARM hypervisor, we provide comparative benchmarks in real hardware, mainly Raspberry Pi. We chose Raspberry PI for the following reasons:

1. PI processor's architecture is ARMv8 and has a Hypervisor and TrustZone.
2. Raspberry Pi is abundant with software and therefore we could compare many available technologies.
3. PI is cheap. Thus, our technology may be compared and extended by others wishing to test our technologies.
4. Some researches that we extended, for instance, C-FLAT, was implemented in Raspberry Pi. This eased the integration as well as we were able to compare our solution to the original solution.
5. It is easy to connect sensors and logical analyzers to the PI to prove hard real-time.
6. Raspberry PI is also presented as hardware that does not fully comply with TrustZone requirements, and as such, we presented a DMA attack on op-tee on the PI.

2 ARM

This section discusses ARM's history, the operating systems available in ARM, the security solutions and real-time technologies.

2.1 Background

ARM is an acronym for Advanced RISC (Reduced Instruction set Computing) Machine. RISC processors requires less transistors than CISC (Complex Instruction Set Computing). This reduce costs, reduces power consumption, and generates less heat. These features are appealing for low-power devices. Therefore, it is not surprising that smartphones embed an ARM soc. The appearance of the Apple iPhone and Google's Android boosted ARM immensely (Figure 1). The smartphone industry, and with it ARM cores, leaped around 2010, but with it, so did the security risks. Vendors were forced to invest a lot more resources in securing their phones.

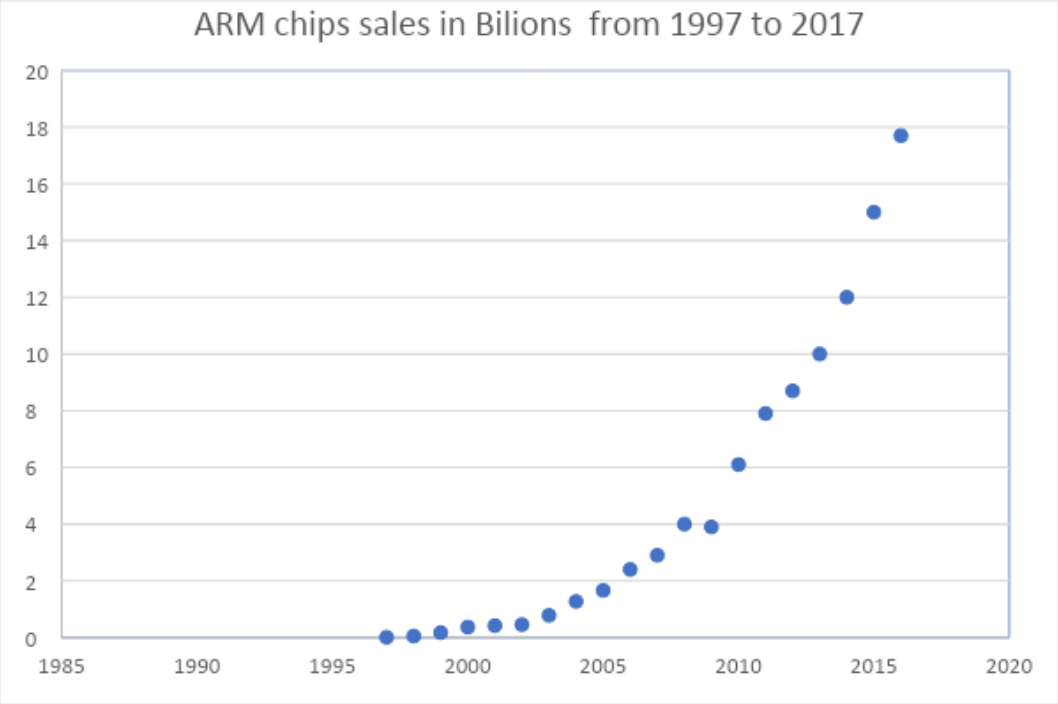


FIGURE 1 ARM incline ©Wikipedia

ARM architecture starts from version ARMv1 and extends to ARMv8. These architectures have models, such as the Cortex family model. While models vary in their speed, the architectures differ in their features. For instance, ARMv6 (ARMv6KZ) is first to have TrustZone, ARMv7-a is the first architecture with a hypervisor, and ARMv8 is the first to support 64bit. This abundance requires software to utilize it, programming languages, operating systems, compilers, and so on.

2.2 Operating Systems in ARM

Prior to the appearance of the smartphone, ARM was mainly used in embedded devices. Prominent yet small operating systems in the embedded industry are FreeRTOS , VxWorks Hambarde et al. (2014), Zephyr Kim and Shin (2018), and others. Symbian Hall and Anderson (2009), another prominent operating system used to run Nokia phones, was running on ARM. Embedded devices tend to serve a single purpose, and therefore do not require a GPOS (General Operating System). One notable exception to the single-purpose characteristics is the Blackberry (1999) Allen et al. (2010) OS. BlackBerry devices used a proprietary Operating system named BlackBerry OS. With the appearance of Android and iOS, Blackberry lost its dominance.
It is quite understandable that Android and iPhone OS (IoS) are the most common operating systems using ARM cores available today. These operating systems extended ARM to execute software that hardly, if at all, existed prior to their

appearance, for instance, Java and Python, frameworks such QT (a multiplatform platform), Unity (games platform), and many others. Other well-known Operating Systems are Windows for ARM, Linux, and FreeBSD.

2.3 Security in ARM

Android, IoS, Linux and Microsoft Windows on ARM are widely common as general purpose operating systems, and therefore are prone to be attacked. Each operating system provides its own techniques to protect itself. Windows provides Windows Defender, and Linux and Android use SELinux. A different challenge emerges if the attacker has supervisor or an admin (root) access to the device. Obtaining a superuser access to a device gives the attacker the ability to disable the defense software. To cope with this challenge, starting from ARMv6, ARM added TrustZone Winter (2008) (Figure 2) to its architecture and Intel added SGX Costan and Devadas (2016).

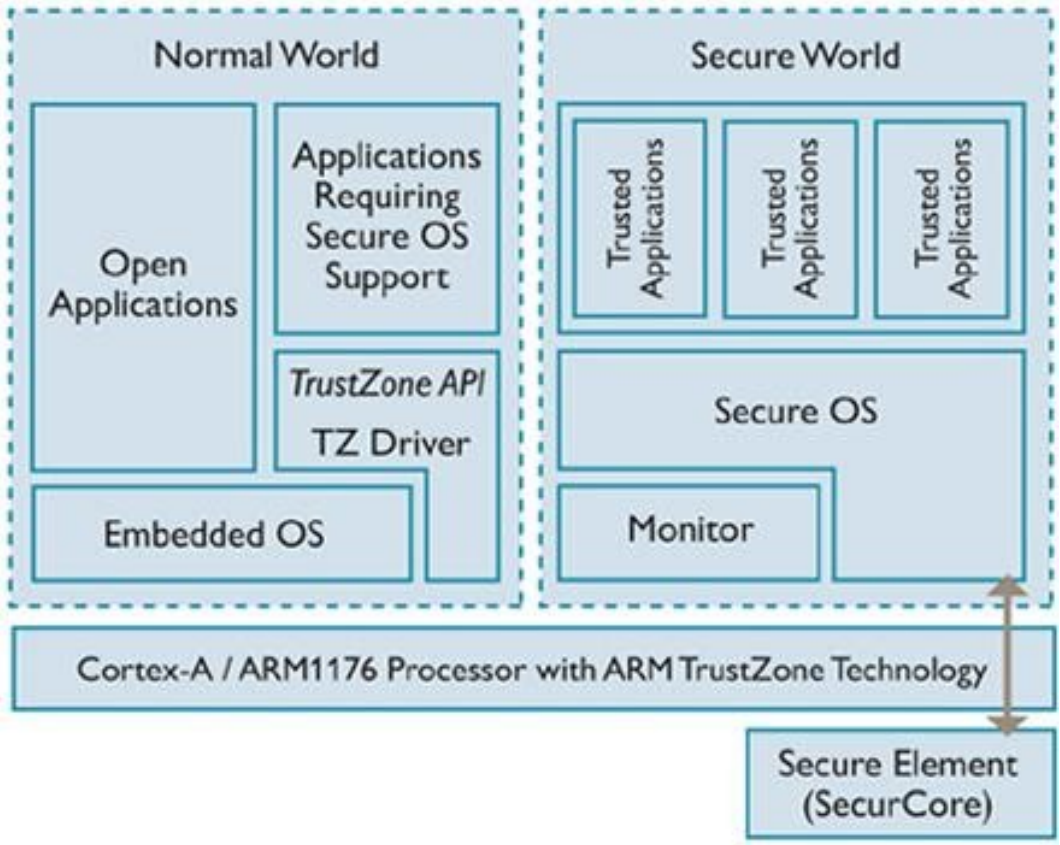


FIGURE 2 ARMv8 Architecture ©ARM

2.3.1 TrustZone

TrustZone is meant to provide a Trusted Execution Environment (acronym TEE) separated from the GPOS (often in the context of TEE, a GPOS is referred to as

Rich OS, or REE, Rich Execution Environment). An open-source software application for TrustZone is Linaro's OP-TEE Linaro (2020). Known closed-source software is Qualcomm's QSEE, used in Qualcomm SOMs.
TrustZone is implemented as a vector of code. The SMC (Secure Monitor Call) command is used to enter the TrustZone (Figure 3) and ERET (Exception return) to exit back.

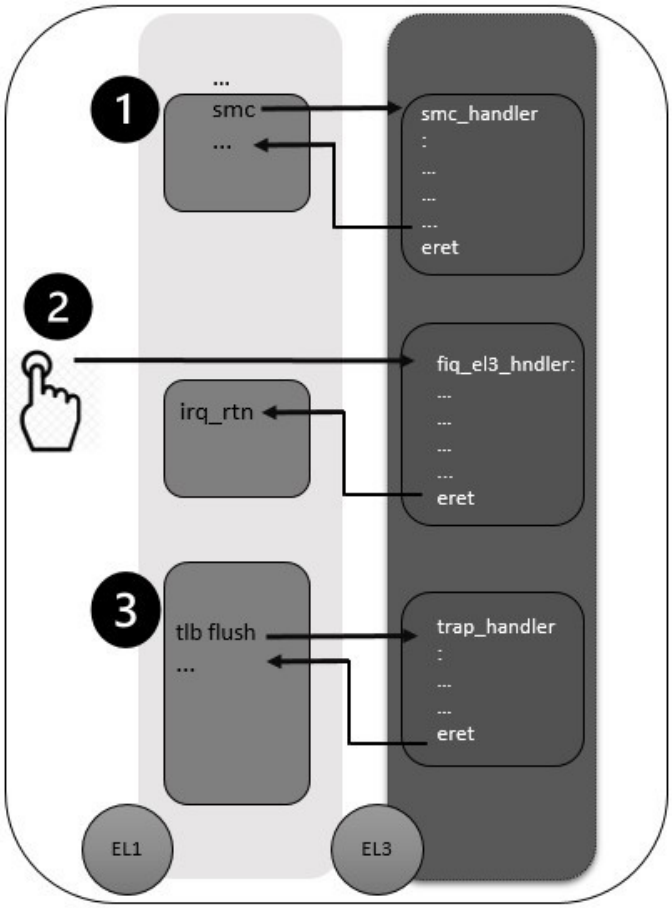


FIGURE 3 ARMv8 TrustZone Interface

Figure 3 depicts four exception levels (EL0,EL1,EL2 and EL3). There are three ways to enter TrustZone:

1. By an SMC. Only from EL1 or EL2.
2. By routing interrupts. Only from EL1 or EL2.
3. Through a programmed trap. i.e., it is possible to define that certain opcodes are trapped to Trustzone. For example, it is possible to trap any access to the TLB, or debug registers to TrustZone. From EL0, EL1, and EL2.

FIQ is a fast interrupt, and it is prioritized over the IRQ (a regular interrupt). Interrupts in ARM are classified as follows:

Secured Interrupt	Group level	FIQ	IRQ
Secured	Group 0	Yes	No
Secured	Group 1	Yes	Yes
Not Secured	Group 1	Yes	Yes

In general, the ARM processor may be in a secure state or a non-secure state. Physical addresses are tagged with NS (non-secure), also referred to as NP (Normal Physical) bit, or SP (Secure Physical) bit. As an example, the address 0x400000 may be two distinct addresses, SP:0x400000 or NP:0x400000, which translates to two distinct physical addresses.

This distinction between the exception levels enables activities such as cache or TLB operations in a non-secure state to affect only non-secure addresses, while in a secure state, it may access and affect all addresses. Figure 4 depicts RAM partitioning to secure regions and non-secure regions. The red areas are secured RAM regions that can be physically accessed by the RICH OS.

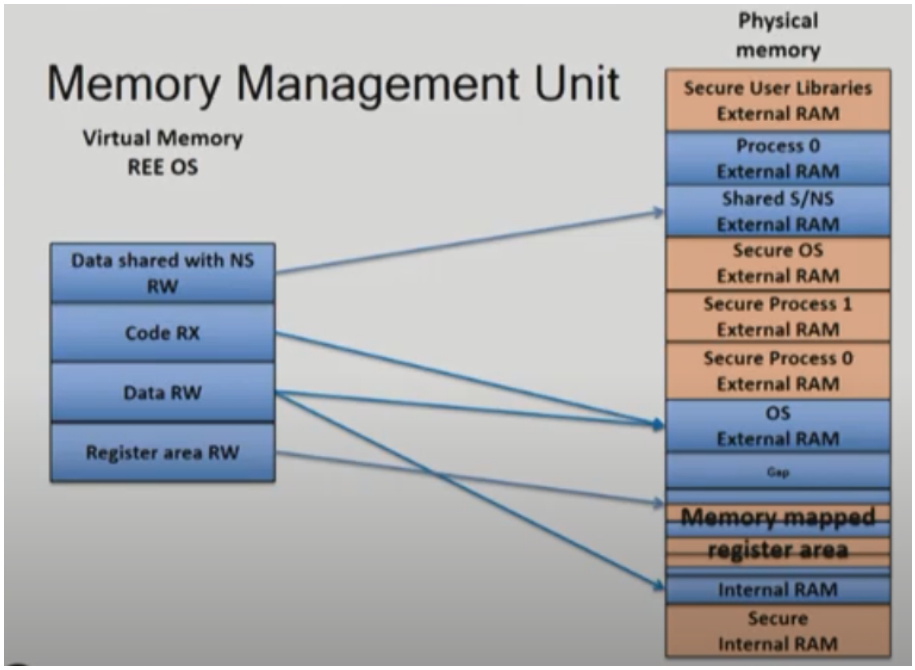


FIGURE 4 ARMv8 MMU ©ARM

ARM's TZPC (acronym for TrustZone Protection Controller) depicted in Figure 5, is a hardware controller. It provides a software interface to set up memory areas as secure or non-secure. In the TrustZone glossary, AMBA is an Advanced Microcontroller Bus Architecture (AMBA) and APB means Advanced Peripheral Bus (APB) protocol specification. To achieve maximal protection, it is essential that the vendor follow the APB specifications.

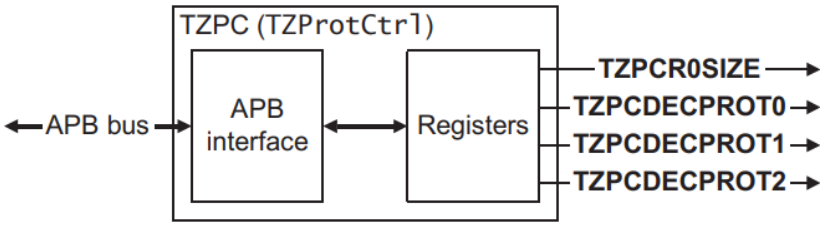


FIGURE 5 ARMv8 Simplified TrustZone Controller ©ARM

In addition to cache and TLB, when in secure mode, some of the system registers are banked. Banked registers mean that there are two copies of these registers, one for the secure world and one copy for the non-secure world.

2.3.2 Cybersecurity challenges

Cybersecurity deals among other things, with computer forensics, trusted execution environment (TEE), Malware prevention and detection, CFI, et cetera.
Malware is software designed to harm the computer. Malware comes in forms such as Trojan horses, adware, spyware, Rabbit, self replication program aimed to slow down the computer, and so on. Computer forensics is a science that examines the computer's media, storage, or volatile memory in search of malware. Forensics tools analyze the media and recover information that points to the existence of viruses. Forensics is performed in two main stages: acquiring the data and analyzing the data. TEE is a separate execution space in the processor. It isolates the code and the data from the GPOS or any other execution spaces in the processor, thereby protecting software from being reverse engineered. Control Flow Inspection (CFI) is a technology for preventing malware from redirecting a flow of a program to perform an attack.

2.3.2.1 Malware challenges

Malware attacks in many forms. Malware might victimize by exploit kits. A vulnerability is a software weakness that can be exploited. Exploit kits are toolkits that scan the software for vulnerabilities, and when a vulnerability is found, the kit can inject malware into the computer. Common malware are:

- Adware is an unwanted advertisement. It is software that automatically generates online advertisements and creates revenue for its creator.
- Malvertising (malicious advertising), is an exploitation injected into an advertisement. It uses the ad platform to spread.
- "Man in Middle" attack is when the attacker secretly listens to the communication of two parties or more and may even choose to alter it. For example, an attacker scans an unsecured router (usually wireless networks) and guesses the router's password. Once it gains this password, it will inject software between the user (the web browser, for example) and the website

and collect data. Using a "Man in the Middle" attack, the attacker can inject data into the user's computer.

2.3.2.2 Computer Forensics Challenges

Computer forensics is the process of extracting information from a raw memory dump. A memory dump is a snapshot of the RAM. Forensics, main challenges are:

- Acquiring a coherent image of the memory (atomicity) Kiperberg et al. (2019a).
- Analyzing the memory image to discover attacks or anomalies.

Finding anomalies is difficult in many cases because malware may camouflage itself and its activities. Coherency of the image is influenced greatly by the way it was acquired. Atomicity of the images requires that the memory not change while it is being acquired. If the process of acquiring the image is performed while the operating system runs, however it is expected that some pages' content will change while acquiring it. Figure 6 demonstrates the challenge of coherent memory acquisition. The longer the duration of the acquisition, the less likely it is coherent. In Linux, LiME is a driver that acquires the memory while the OS runs. The Volatility Dave et al. (2014) memory acquisition framework uses the LiME driver.

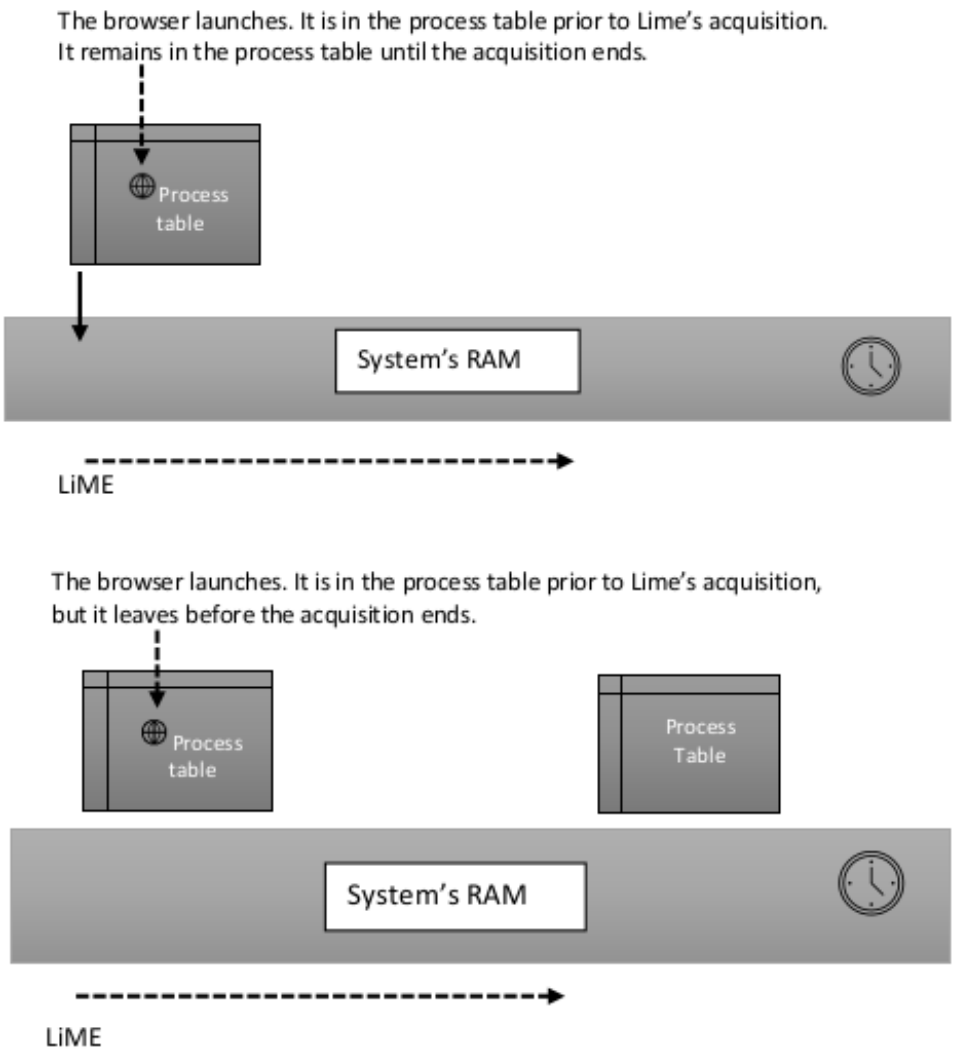


FIGURE 6 The Incoherent Image Problem

To tackle the incoherent image problem, acquisition tools tend to acquire the memory as fast as possible or freeze a virtual machine and grab a RAM snapshot. The Volatility framework also handles COW (Copy-On-Write).

2.3.2.3 Trusted Execution challenges

TEE (Trusted Execution Environment) usually comes in the form of isolated privileged areas in the processor and RAM. In Intel processors, these areas are referred to as enclaves, which are part of Intel's SGX (Software Guard Extensions) technology. In ARM this area is known as TrustZone, and IBM introduced zACI for its zSeries servers. Intel's SGX and TrustZone are essentially responsible for attesting the authenticity of the RICH operating system, its authenticity, and trust-ability. TrustZone and Intel's SGX, by their hardware design, are expected to run safe and trusted operating systems that cannot be compromised.
The various TEEs are widely researched, and in many cases are penetrated via sophisticated attacks; Intel SGX was attacked by SgxPectre Chen et al. (2019), and

ARM was attacked by the manipulating PMU (performance management unit) or the debug registers. Ning and Zhang (2019) et al. show that because the ARM debugging model requires no physical access, a low-privilege host can use ARM debugging features to gain read/write access to TrustZone secure world. This allows a low-privilege host to initiate a debug session with a high-privilege target using these debugging features.
Ning and Zhang (2019) et al. used the ARM debugging features to leak private keys from the Secure World, thus compromising ARM TrustZone security. Ning and Zhang (2019) et al. suggested that ARM should add restrictions in the inter-processor debugging model to enforce permission between host and target.
Spisak (2016) et. al. describe another processor feature-based attack using ARM CoreSight debug features. Spisak (2016) et. al. leverage ARM's PMU to create a rootkit that cannot be detected by the kernel monitor because it does not change the kernel syscall but rather attaches through the PMU to any syscall. Thus, every syscall will raise a PMU event, and the rootkit will be able to modify the input and output data of the syscall. This attack is possible due to a hardware implementation bug of the debug signal authorization that enables debug features in the hardware.
DMA attacks on TrustZone are possible when vendors do not fully comply with the architecture of the processor and "cut corners" to reduce costs.
Other attacks are directed on the cache lines that take advantage of the coherence protocol in a multiprocessor computer, or monitor cache activity caused within the ARM TrustZone from the normal world.

2.3.2.4 CFI challenges

CFI is a set of techniques aimed to protect against attacks that redirect a flow of a program's execution and force the program to execute malicious code. The ROP (Return Oriented Programming) vulnerability enables arbitrary code execution. An attacker controls the call stack, so that it manipulates the return address of a function to hijack a program. There is an arsenal of defenses against ROP attacks, to name a few; stack guard, ASLR (Address space layout randomization), et cetera.
A big challenge in CFI is performance. CFI carries with it the penalty of an additional execution time. Therefore, it is usually preferred to use hardware for CFI, for example PAM ARMv8.3-PAuth adds PAC (Pointer Authentication Code) Liljestrand et al. (2019). PAC mainly targets pointer substitution attacks and arbitrary memory reads and writes. PAC validates whether the target of an indirect branch is correct. Indirect branch, also called forward-edge, can be, for example, a function pointer. PAC also protects against backward-edge attacks (ROP attacks).
However, ARMv8.3 is scarcely available at the time of writing. In addition, PAC does not protect from Data Execution Prevention (DEP) attacks Liljestrand et al. (2019). Though hardware solutions are an excellent technique to safeguard native code, it does not protect interpreters (Python) or Just-in-Time Compilation

(JIT). These technologies make it easier to inject data into a system, which are then interpreted as instructions.

2.3.3 ARM Security Alternatives

2.3.3.1 GlobalPlatform

GlobalPlatform (2011) is an alliance of many mobile device manufacturers. It certifies the standards for mobile devices, including a standard for secure digital services for mobile devices, and is responsible to publicise them. GlobalPlatform acts as the industry standard of the TEE under ARM.

2.3.3.2 General Dynamics OKL4

Originally developed by Open Kernel Labs, the L4 operating system developed by Liedtke Elphinstone and Heiser (2013) is the predecessor of the OKL4 microkernel. The OKL4 that was developed by the Open Kernel Labs, is also maintained and distributed by it. It is a Unix-like operating system. The OKL4 operating system was based on the L4 operating system.

In its earlier architecture, the L4 microkernels family was called L3. As in the L4 kernel, Liedke was also responsible for the development of the L3 microkernels. Liedke created L3 in the 1980s for the i386 architecture, and L3 was deployed mainly in universities. The L3 inter-process communication (IPC) latency was over 100 microseconds. Therefore, Liedtke re-designed and re-implemented L3 IPC, and lessened Liedtke et al. (1997) it significantly.

The Open Kernel Labs named the L3's new design L4. L4 microkernel had evolved and grown to become a family of kernels; to new a few: NICTA, Codezero, L4-embedded, seL4, et cetera. OpenLabs maintains NICTA and renamed it to the OKL4 microkernel.

2.3.3.3 seL4 microkernel

seL4 Klein et al. (2008), a hard real-time operating system, is also based on the L4 microkernels family. seL4 is a microkernel that was implemented by the NICTA group (2006) and by Open Kernel Labs (which was renamed later to GeneralDynamics). It is not as popular as OKL4. One of the strong features of seL4 is that the implementation of the seL4's kernel is correct against its functional specifications. The proof guarantees that the seL4 does not have livelocks, deadlocks, buffer overflows, and arithmetic exceptions. The NICTA group proved seL4 correctness on its C re-implementation. However, though rigorously tested, seL4 is not necessarily bug-free. The implementation has some assumptions of correctness about the compiler, architecture, and C reimplementation. DARPA's High-Assurance Cyber Military Systems (HACMS) program embraced seL4 to provide an operating system to its drones, its autonomous helicopter Boeing AH-6 (unmanned Little bird). DARPA also included seL4 in other Small Business Innovative Research (SBIR) initiatives, which include DornerWorks, Techshot Wearable

Inc, and others.
The basic rule of the L4 kernel design is minimalism. Leidtke (1995) formulated the rule of minimization as follows:
"A concept is tolerated inside the u-microkernel only of moving it outside the kernel, i.e. permitting competing implementation would prevent the implementation of system required functionality."
In other words, only minimal mechanisms and no policy in the kernel. This principle, known also as the no-policy in the kernel, is the core of the L4 microkernel design.
Operating systems tend to inflate over time, as an example, the Linux kernel had grown from a few thousand lines of code(1991) to over 23 million lines of code (2018). In this aspect, microkernels do not resemble kernels, and their LOC (lines of code) tends to remain low over time. As an example, L4's footprint is considerably low and consists of less than 10000 lines of code (2019). Microkernels side effect of the concept of performance and minimization, endeavors abstraction of the hardware, and only a small portion of its code is portable between the various platforms. The abstraction is performed by the higher-level constructs on top of the microkernel.
In L4, interrupts are disabled while the processors execute in kernel mode. The motivation for this approach was to increases performance and eases the formal verification. Another facet of minimization is the memory resource management. seL4's memory manager is located in user space.
The context switch is another common performance penalty operating systems must cope. The L4 approaches this problem by applying the technique of direct process switch. In a direct process switch, the kernel tries to avoid the scheduler as much as possible. When a thread is pre-empted, the kernel chooses the first available thread.
Presently, seL4 is avaialble on ARMv7 and ARMv6 and x86 32bit. It also support SMP. In seL4's website, there is a short list of platforms available to seL4, which implies the reason for its small market portion.

2.3.3.4 Google Trusty TEE

Developed and maintained by Google, Trusty is a secure operating system for the Android operating system. It is open-source and part of the AOSP (Android Open Source Project). Similar to other ARM TEE technologies, Trusty executes in EL3, i.e., it utilizes ARM's TrustZone, thereby creates a distinction between the normal world and the trusted world. Trusty consists of:

- The Trusty Kernel. This kernel is part of Android's LK (Little Kernel). LK is Android's boot loader.
- A Linux kernel driver that acts as a bridge to Trusty.
- An abstraction layer for Google's applications. This is userspace library (shared object).

Trusty is available in Intel and ARM processors.

2.3.3.5 Linaro OP-TEE

OP-TEE operating system for ARMv8 and ARMv7, is a joint effort of Linaro's security team and STMicroelectronics. OP-TEE is available under the BSD 2-clause license, is an open-source project. Some of its kernel parts are versioned under the GPLv2 license. The GlobalPlatform specifications, both the TEE Internal Core API and the TEE Client API implementations were applied to OP-TEE.
We evaluated OP-TEE's performance overhead on a running Linux system. Our benchmarks were performed on a Raspberry PI3. Similar to Müller et al. (2019), the evaluation showed that OP-TEE does not affect the REE performance. As part of Linaro, OP-TEE has a large community of developers and users, and is well documented with examples. OP-TEE consists of:

1. A memory management component, interrupt handling component and so on. The upper layer of the kernel implements a HAL, mainly to provide support to the various platforms. OP-TEE is capable of running user-space applications and kernel space. Userspace applications in the secure world are typically referred to as Trusted Applications or in short TAs. The TAs abide by the GlobalPlatform specifications. This API enables the secured kernel to serve the TA securely.
2. An open-source, Linux kernel driver that handles data transitions between the secure and non-secure worlds.
3. Software Layers that make the transition from the secure monitor to the secured op-tee kernel and from there to the secured userspace, and back.

As ARM virtualization technology grows so does the need to secure it grows. Virtualization is supported on OP-TEE. i.e., it is possible to invoke a TA from different VMs. However, OP-TEE is unable to translate a GPA (Guest Physical Address), and thus requires a hypervisor intervention. The TEE mediator is the additional component of the hypervisor that performs the GPA translation for OP-TEE.

2.3.3.6 Kinibi

Another interesting operating system for Android is Kinibi, provided by TrustTonic. Kinibi, a closed source operating system is common in smartphone (Samsung). Kinibi main feature are:

- Data encryption
- Secure access through its Trusted Execution Enviroment (TEE) to the phone peripherals. for example: NFC, touch screen, finger print reader et cetera.
- Device authentication
- Safe Code Execution and data security.

The verification of Kinibi is done by a chain of trust, initiated by the bootloader in each device boot. Furthermore, since Kinibi safely accesses I/O devices, it can

provide safe access to the network device. Thus, a trusted application may invoke remote services securely.
Another market TrustTonic approaches are the automotive industry. Here, TrustTonic addresses application overlapping attacks, data leakage attacks, and application re-packaging attacks. Application overlapping attack is a method in which an attacker steals sensitive data by re-routing the I/O path, for example, when the user enters a password. Repacking of an application is a technique of modifying a code to steal sensitive data. For instance, printing sensitive information.
Kinibi is compliant with GlobalPlatform API specifications and offers an SDK that aids the construction of a trusted application.

2.3.3.7 Xen

Xen was developed by Ian Pratt at Cambridge university and announced in 2003. In the glossary of Xen, a domain is a virtual machine. The first domain is referred to as Dom0, and it is a Linux system or BSD. Dom0 has access to the entire machine's hardware, and must run before the execution of any virtual machine. The virtual machines run on top of the other domains. A virtual machine has no access to the underlying hardware, and therefore it is called an unprivileged domain (DomU). The Linux kernel (or BSD) provides services to Dom0. The communication between Dom0 and DomU is done by Xen's event channel. By implementing virtual interrupts, timers, communication between guests and MMU virtualization Xen virtualizes machines.
Any virtualized event from Dom0 is passed to the event channel. Xen supports both full virtualization and para-virtualization. Pure virtualization is done by QEMU. Para-virtualized events are passed through the event channel to the para-virtualized guests. As noted earlier, Dom0 runs Linux because Linux has wide hardware support, and has abundant software. Xen's management tools, called toolstack, are used to control the guests. Xen's is a type-1 hypervisor, and boots from the bootloader and then loads the para-virtualized host.
I/O virtualization usually comes with a performance penalty, and Xen is not an exception. Virtualized interrupts and I/O accesses are delegated to Dom0 from the Xen guests. An interrupt that occurs while DomU executes is recorded and is served only when Dom0 gets the processor.
Xen is available in ARM and x86 and runs on SMP and UP, and is licensed under GPL.

2.3.3.8 Xvisor

Announced in April 2012, Xvisor Patel et al. (2015) is a type 1 hypervisor. Xvisor is a monolithic hypervisor. It is agnostic to the guest's internal structure. Xvisor supports ARM 32bit, 64bit, and x86. It also supports multiple processors (SMP) computers and single-processor computers. In this sense, a guest can utilize two or more processors. Xvisor hypervisor controls the computer periphery and provides a small operating system. Device passthrough usually provides the means

to use the periphery, but Xvisor also supports device emulation. As an operating system, Xvisor provides memory management, scheduler, threading, and so on. However, Xvisor is not POSIX compliant, and many POSIX standards were not implemented to its kernel. A notable example is the lack of processes.
An important feature of Xvisor is the possibility to perform IPC between guests. This IPC referred to as an aliased region is a GPA (guest physical address) shared between the guests. In Xvisor, a processor is called vCPU (Virtual CPU). Xvisor defines two types of vCPUs:

1. Normal vCPU. This processor serves guests.
2. Orphan vCPU. This processor belongs to the hypervisor.

Xvisor footprint is approximately 10MB, and therefore it is not small. As it is a type-1 hypervisor, some modifications are required to the bootloader.
Xvisor targets the infotainment market, mainly in the automotive world. Xvisor is licensed under GPL.

2.3.3.9 QSEE

Qualcomm Secure Execution Environment, referred to as QSEE, was developed by Qualcomm from scratch in 2015. In the past, it was based on OKL4, but GeneralDynamics and Qualcomm failed to reach a licensing agreement. QSEE is a closed source operating system and is widely spread in the mobile industry.
QSEE is closed source (and Qualcomm does not provide source code licenses).

2.4 Real-time

A real-time operating system (RTOS) is measured by its predictable responsiveness to an event. A real-time operating system guarantees a response within a deadline. A key character of real-time is jitter. Jitter measures the level of consistency that concerns the amount of time it takes to perform a computation. The usefulness of the result is regarded by some as a way to categorize real-time; hard real-time and soft real-time. Hard real-time means that computations that miss their deadline fail the system, compared to soft real-time, for which late computations degrade the system but do not fail it. A good example of hard real time is a navigation systems, where high jitter may end with a catastrophe. An example of soft real-time is video or audio, where a glitch in the screen or a crack in the sound is not considered harmful.

2.4.1 Real-time Operating Systems in ARM

ARM Real-time technologies vary in their architecture. Some vendors provide a complete operating system, such as VxWorks (653), FreeRTOS or Zephyr. Some offer a microkernel (the minimal software needed to implement an OS) such as seL4, and some offer an extension to a GPOS, such as Linux RT_PREEMPT or

microvisors (a microkernel within a hypervisor). In the microcontroller arena, vendors offer a bare metal solution, such as CubeMX for the STM32 micro controllers (Cortex M4) family.
Figure 7 compares seL4, RT PREEMPT Rostedt and Hart (2007), and Xvisor. seL4 provides hard real-time with a jitter up to 5 us in a 1 ms interval.

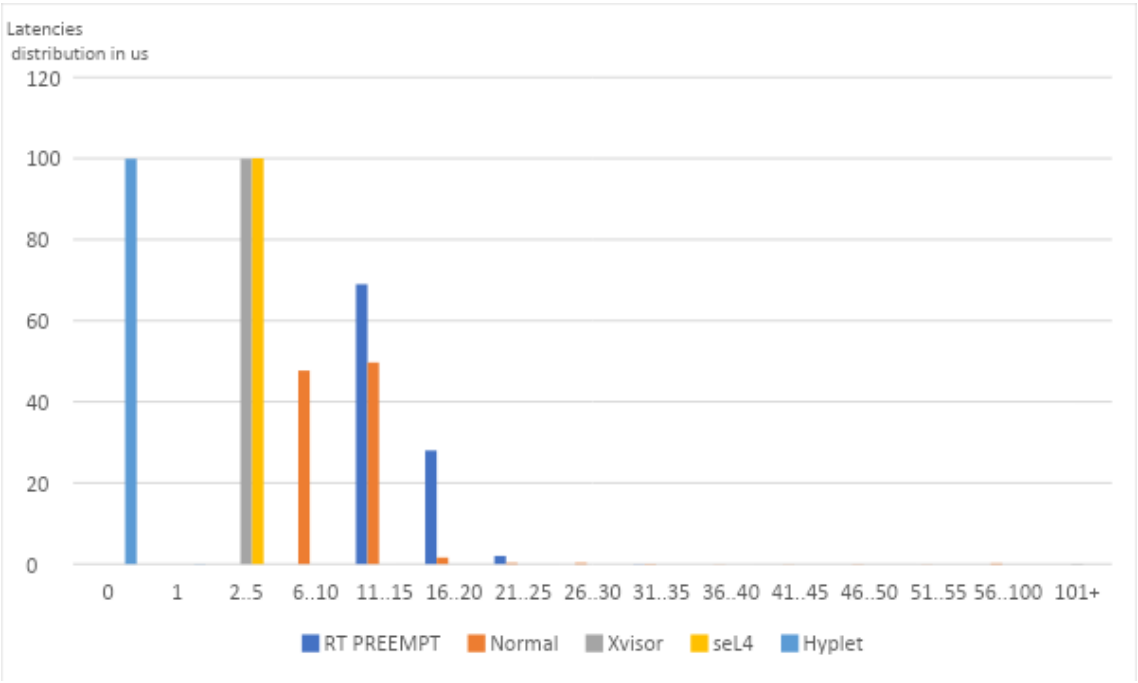


FIGURE 7 Raspberry PI3 timer latencies, 1 ms interval

2.4.2 Real-time challenges

As the need for low-power devices that require some form of soft real-time or hard real-time grew, ARM operating systems, such as FreeRTOS Guan et al. (2016) and Zephyr Kim and Shin (2018), became more popular. As the industry grew and the requirements from the operating system increased, a simple operating system was not enough. A GPOS is required to perform all these tasks. For example, many devices today require GUI in addition to real-time, and some require support for multiple programming languages, web browsers and so on. Linux, as an open-source abundant operating system suits this niche very well.
Linux RT PREEMPT is considered by some as a soft real-time kernel that runs native Linux distributions such as Debian. Linux RT PREEMPT changes the way the Linux kernel handles interrupts. In simple terms, interrupts become threads that can be preempted by any user space thread. Other known open-source real-time operating systems for Linux are Xenomai Gerum (2004) and RTAI Mantegazza et al. (2000). Both technologies employ a microkernel architecture, meaning that the Linux kernel is merely a background task. Both technologies run on most processor architectures, e.g. x86, ARMv7, Power ISA et cetera. According to Barham et al. (2003), these two technologies perform somewhat the same with RTAI being a bit faster. seL4, discussed earlier, offers real-time Blackham et al. (2011) and

security Sewell et al. (2011). The weakness of seL4 is, however, its assimilation to existing hardware. It requires mastering CAmkES Kuz et al. (2007), a software component for microkernel-based embedded systems and a framework to build an operating system.
Another evolution of the L4 operating systems family is the L4Linux, on top of the L4Re Lackorzynski et al. (2016) microkernel. This technology can execute ordinary Linux threads separately from the Linux operating system, relieving them to the Linux kernel heuristics. Lackorzynski et al. (2016) et al. extended L4Linux to execute user space interrupts directly, by routing them from the microkernel to the L4 Linux kernel. L4Linux implements threading APIs similar to Linux, to achieve seamless traversing from Linux to the L4 kernel. L4Linux was adopted specifically to access user-space threads by reusing its address space; it requires virtual unplugging processors from Linux. Virtual unplugging of a processor is Linux's ability to remove any kernel activity from an active processor.

Another challenge in real-time systems is security. Security techniques tend to slow down software operations, thereby increasing jitter.

3 HYPERVISORS

3.1 Overview

A hypervisor is software or hardware or both that creates and runs virtual machines. Virtualization creates pseudo devices, such as virtual processors, virtual memory, virtual network cards et cetera. Modern virtualization requires a processor's extension called a hypervisor. A hypervisor supervises supervisors, which are kernels. Popek Popek and Goldberg (1974) classified hypervisors as follows

- Type-1 hypervisor
  This is a hypervisor that runs directly on the hosting machine without the intervention of the operating system. Examples are Xvisor for the ARM platform and Microsoft Hyper-V.
- Type-2 hypervisor
  This hypervisor runs from the hosting operating system, many times as a loadable driver. Examples are Linux KVM, VMplayer, and QEMU.

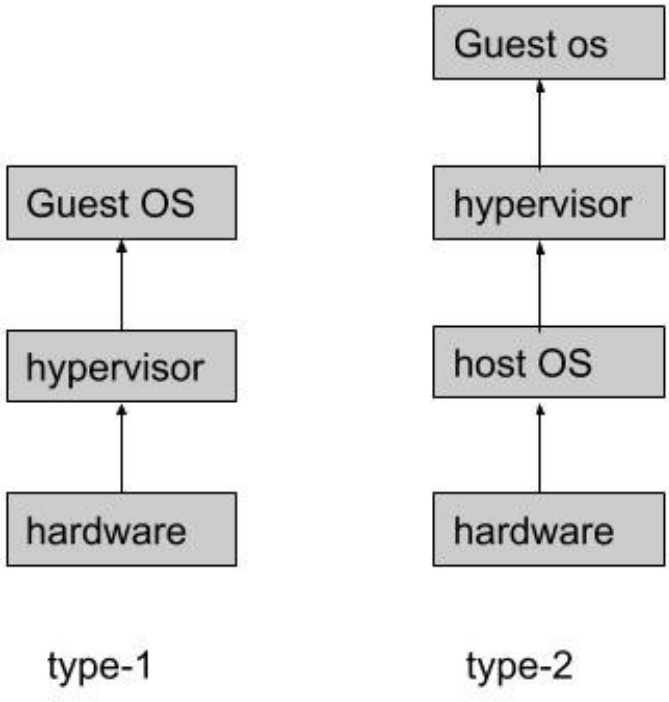


FIGURE 8 Hypervisor Types

In a type-2 hypervisor, a guest OS is the operating system, which accesses only virtualized resources while a host operating system accesses the real hardware. The software that manages the virtual machines is called a VMM.
Other known classifications for virtualization are full virtualization and paravirtualization. In paravirtualization, the guest OS is aware of the hypervisor. The awareness is expressed by the guest OS asking the hypervisor to access hardware resources. This is in contrast to full virtualization, where the guest operating system is not aware of the hypervisor. As a result, Paravirtualization is considered supreme in its performance. However, this depends on the hypervisor software and the guest software. A known example of paravirtualization is Xen.
A different hypervisor classification is by its goal. As explained earlier, the main goal of hypervisors was to reduce the costs by virtualizing the hardware, but over time other needs were required; for instance, a real-time operating system on top of a hypervisor (microvisor) or securing the GPOS by wrapping it with a hypervisor. This type of hypervisor does not virtualize the hardware but allows the guest OS to access it almost directly. This hypervisor is called a thin hypervisor. A thin hypervisor provides better security Algawi et al. (2019) and better performance Baryshnikov (2016) to the standard host machine.

3.2 ARM Virtualization

ARM security is achieved through its exception levels (EL) architecture Penneman et al. (2013) . In ARMv7, ARM introduced the concept of a secure world and

a non-secure world, through the implementation of TrustZone and, starting from ARMv7-a, ARM presents the following permission (exception) levels.

- **EL0** refers to user-space. EL0 is analogous to ring 3 on the x86 platform.
- **EL1** refers to the operating system. EL1 is analogous to ring 0 on the x86 platform.
- **EL2** refers to the hypervisor. EL2 is analogous to ring -1 or real mode on the x86 platform.
- **EL3** refers to TrustZone. It is a special security mode that can monitor the ARM processor and may run a real-time security OS. There is no direct x86 analogous mode. Intel's ME or SMM are related concepts on the x86 platform.

The first exception level, the second and third have (Figure 9) their special-purpose registers and can access these registers at the higher but not lower levels. The general-purpose registers are shared. Thus, moving to a different exception level does not require the expensive context switch associated with the x86 architecture.

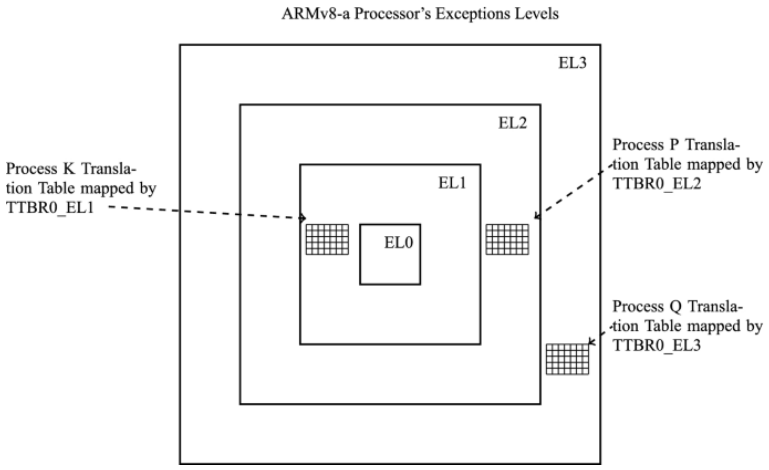


FIGURE 9 Hypervisor Exception Levels

To understand how virtualization works in ARM, we first must explain how memory is virtualized. Figure 8 depicts how memory translation takes place. First, the hypervisor sets up an additional translation table, called a stage 2 table, then it programs the processor to move from a single stage translation scheme to a two-stage translation scheme. In a two-stage translation scheme, each physical address accessed is from one table to a second table (Figure 10), thereby fooling the guest OS to detect a physical address when it is a virtual address.
This technology is called SLAT (Second Level Translation Table). In ARM, SLAT is called IPA, Intermediate Physical Address; in Intel it is called EPT (Extended Page Table).

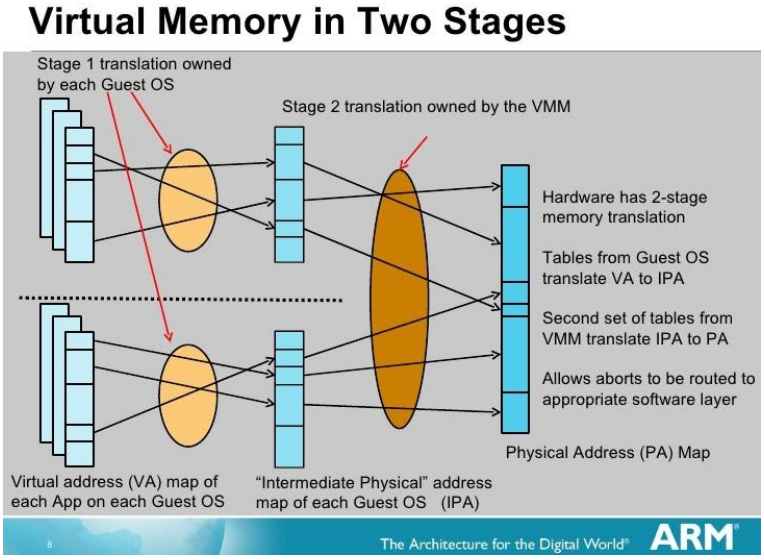


FIGURE 10 SLAT in ARM ©Pratt

Interrupt routing from a device to the guest operating system is done by programming the hypervisor to propagate the interrupts to a certain guest operating system. A hypervisor is a vector that is accessed as follows:

1. Through an HVC (Hypervisor Call)
2. By routing interrupts
3. Through a programmed trap
   A command that is programmed to be trapped into the hypervisor. This includes commands such as access to the MMU translation tables, TLB, debug registers, and so on.

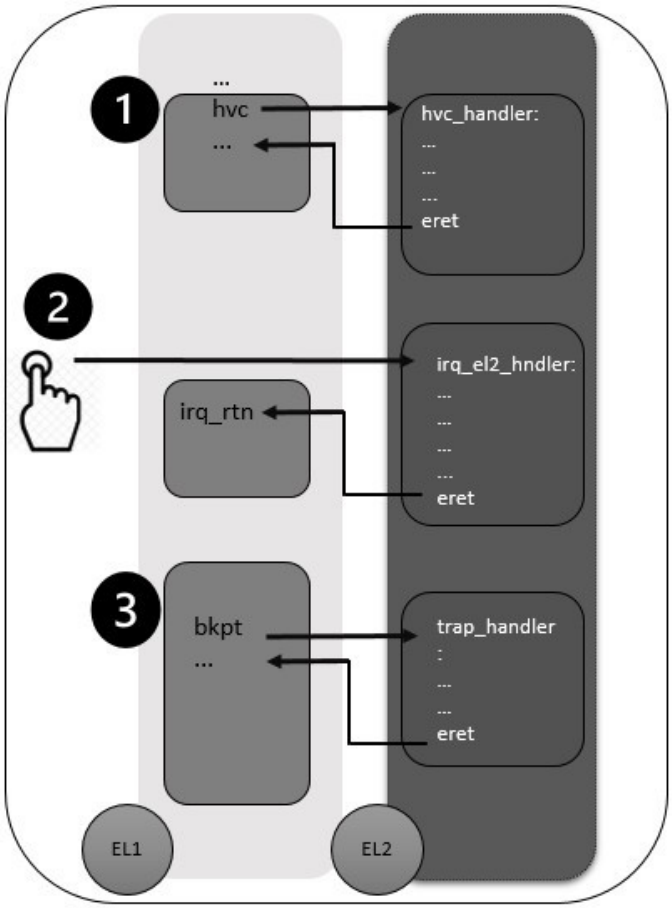


FIGURE 11 ARMv8 Hypervisor Interface

Figure 11 (similar to Figure 3) depicts the ways to access the hypervisor exception vector Dall and Nieh (2014a). Any of the above methods shifts the processor to privilege level 2. In ARM, when the processor moves between exception levels, in addition to the special registers it may or may not access, each exception level (EL1/0, EL2, and EL3) has an independent translation table.

Therefore, Virtual address X in EL1 does not necessarily map to the physical address of EL2 to EL3. In fact, address X may not be mapped at all in other exception levels as depicted in Figure 12. This separation does not refer to the virtualization, but to the address space of the hypervisor or the TrustZone. In addition, there is no requirement to have the translation table non-accessible from other exception levels. This means, for example, that EL1 may construct EL2 or EL3 page tables and vice versa. Figure 12 depicts this sort of mapping. EL1 has access to the EL2 memory table. This technique is used by Linux KVM Dall and Nieh (2014b) in ARM.

The boxes represent the page tables of EL1/EL0 and EL2. These page tables are distinct. Here, however, EL2 page tables are controlled by EL1 kernel. The small low box in EL1 page table contains the page table of EL2

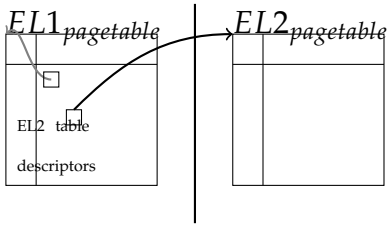


FIGURE 12 Translation table crosses Exception Levels

Starting from ARMv8-1a, ARMv-8 translation tables support both user space and kernel space addresses (Figure 14). This feature is referred to as Virtual Host Extension (documentation arm (2021a)). In previous architectures (Figure 13), the hypervisor could only provide user space addresses. A user space address in ARM is one whose 12 most significant bits are zeroes, while in a kernel space these bits are filled with ones.

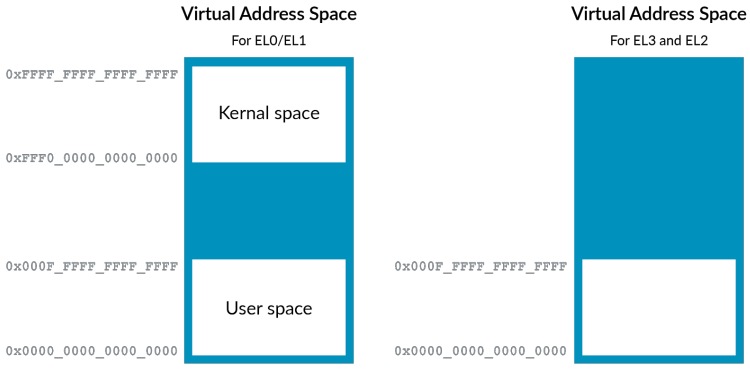


FIGURE 13 Address space prior to VHE ©ARM documentation arm (2021a)

Therefore, in processors that do not support VHE, it is only possible to run user space code without changing its mapping, while kernel code requires some sort of shifting. For example, if we want to map the code of a user-space program (Figure 15) we simply create a virtual address of 400,000 and access this page without any special preparations. We refer to this access as native hypervisor access.

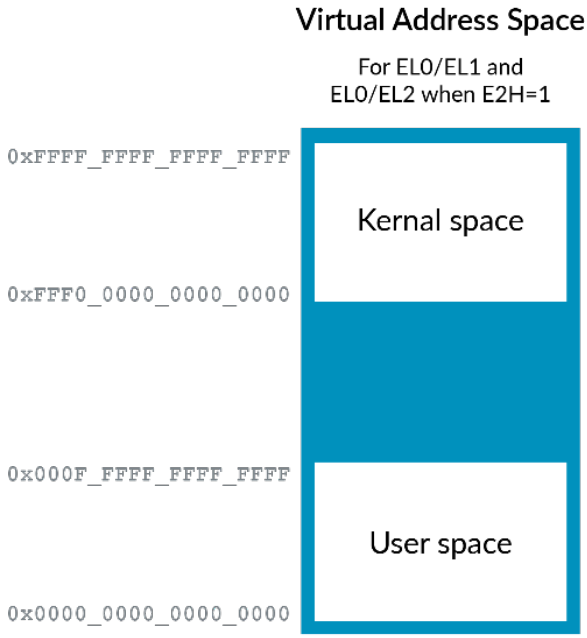


FIGURE 14 Address space with VHE ©ARM documentation arm (2021a)

```
400610:  func:
400614:  stp x16,x30, [sp,#-16]!
400618:  adrp x16, 0x41161c
40061c:  ldr x0, [sp,#8]
400620:  add x16, x16, 0xba8
400624:  br x17
400628:  ret
```

FIGURE 15 func mapped to EL2 and EL0

When we want to access kernel data structures, we must make some preparation for the above. Linux KVM does this by adding to the kernel's virtual address user space page boundary. With VHE this shift technique is not required any more and accessing kernel code and data requires a simple mapping. The most common virtualization software in ARM is KVM. KVM (Kernel Virtual Machine) is a type-2 hypervisor, and an open-source software used by many other projects. In addition to KVM, Xvisor, a type-1 hypervisor, is another open-source virtualization technology in ARM. Another example of ARM hypervisor technology is OKL4, but it is considered obsolete.

3.3 Security of thin hypervisors

This section discusses the use of hypervisors to provide security. Usually, ARM uses TrustZone as the technique for security, and many times the hypervisor and

TrustZone behave the same. The main difference is that in some cases TrustZone software is not accessible for researchers, and therefore it is not easy to employ security and real-time techniques on top of it. Further, there is no contradiction between TrustZone and the hypervisor; in addition to TrustZone the hypervisor may be employed to provide an extra layer of safety. Here are some important reasons to use the hypervisor:

1. The IPA used for virtual machines is by nature best used in the hypervisor.
2. Virtualization provides the ability to run multiple virtual machines, unlike Trustzone, thereby supporting partitioning of thin hypervisors. Further, ARMv-8.4 provides a new feature that allows to running VM in the secure world.

Naturally, there are many ways to attack a computer. We list here some protection schemes possible by hypervisors.

3.3.1 Translation tables protection

Hypervisors by nature separate the host from the guest. In Type-2 hypervisors, the host constructs the translation table of the hypervisor and the VM. As noted, the hypervisor's (EL2) memory translation table differs from the host's. Therefore, there is a need to protect this table from the host (EL1). To do that, once the hypervisor's MMU is active, and the host is wrapped by the thin hypervisor, the hypervisor can disable access to its table from the host. This is done by setting the hypervisor's translation table as non-accessible in the stage 2 table. Any access from the host to this table must go through the hypervisor. For instance, if the host wishes to perform a mapping in EL2, it must be granted permission by the hypervisor.

3.3.2 Cache and TLB

In ARM, each level of cache line is private and accessible only by higher privileged layers. For instance, EL3 may access EL2 cache lines, so privilege caches may be used to install secret keys and so on. EL2 cache level 0, level 1, and level 2 are privileged caches and cannot be accessed by EL1. Therefore, cache operations cannot tamper with EL2 caches while in EL1. The same applies for TLB operations. ARM tends to use inclusive caches, i.e. some Li cache lines may be included in L1 cache lines. A Last Level Cache (LLC) attack usually works on multicore systems, where program B fills the same cache- level lines of program A, thereby forcing program A to reload its data from RAM (or any other source). However, as noted, accessing privilege caches from low privilege is prohibited and therefore does not pose a threat. In virtualization, VM's are separated into domains. The VMM uses VMID to distinguish between caches of each VM, thereby reducing cache coherency problems and expensive TLB flashes. The VMM also eliminates the security risks that follow these failures.

3.3.3 Secure interrupts

Securing interrupts in HYP mode is possible by routing them into the hypervisor. The hypervisor responsibility regarding this is to map the physical interrupt to the virtual one. Doing so allows the hypervisor to perform some safety checks on the interrupt. The hypervisor for instance may choose to drop the virtual interrupt altogether and process it in HYP mode.

3.3.4 DMA attack

A DMA attack is done by accessing physical memory via a DMA-capable device that bypasses the virtual machine two stage translation regime. ARM-v8 systems come with an SMMU (system memory management unit) that is able to perform a two-stage IO translation in the same manner done by the processor's MMU. Much like the VM translation table, SMMU translation tables need to be constructed by the hypervisor. In cases where there is no SMMU available, or if it is not used, a DMA attack is possible both on the hypervisor and the Trustzone.

3.3.5 CFI attacks

A possible Prevention of Control flow attack using the hypervisor (or TrustZone) consists of several steps.

1. Constructing some expected correct flow data
2. Placing these data somewhere
3. Executing the program
4. Detecting the infected flow
5. Handling the situation

Stage 3 and naturally stage 1 are done in a non-safe environment. Stage 4 is best done in a safe execution environment, otherwise the malware may manipulate the detection itself. Executing the detection in HYP mode (or TrustZone) provides this TEE.

3.3.6 Forensics

A rootkit is a malware that hides and provides continued access to a compromised machine Blunden (2012). Rootkit research is a cat and mouse game on all computing platforms. Researchers develop better forensics to detect rootkits while others develop state-of-the-art rootkits. Despite having completely different authors and purposes, both software applications contained a similar concept of masking their existence by hiding the malware files and the running processes. Live memory acquisition is a tool used by forensics researchers to reverse engineer the malware. Forensics researchers attempt to identify the authors, their goals, and any weakness in the malware itself.
Recently there has been a rise in the use of and research on hypervisors for machine introspection Zaidenberg (2018). Researchers approached the challenges of

acquiring memory on multiprocessor systems, memory randomization by ASLR, the acquired images' coherency.
The Libvmi Payne (2012) library provides introspection services under KVM. Libvmi provides VM- based snapshots and has an integrated Volatility Farmer and Venema (2009) plugin. It was also suggested to use Lguest Russel (2007) or XenBarham et al. (2003) for detection of kernel bugs Khen et al. (2013). Other uses of hypervisors are profiling Menon et al. (2005), Hypertracing Benbachir and Dagenais (2018), and Kernel vulnerabilities Zaidenberg and Khen (2015). Forenvisor Qi et al. (2016) uses the hypervisor to gather and store forensics data on the cloud for later inspection. Kiperberg et al. (2019a) provided a system for atomic memory acquisition and guaranteed atomic access in x86 architecture.

3.4 Real-time Hypervisors

As noted earlier, due to the need to have hard real-time with a general purpose OS, various technologies emerged, one of which is hypervisor partitioning. Embedded Partitioning is a technology that divides a computer into several distinct independent operating systems, for example, VxWorks in tandem with Windows. In many cases, it assigns each core (Figure 16) its own OS, and utilizes a hypervisor to route interrupts to each kernel.

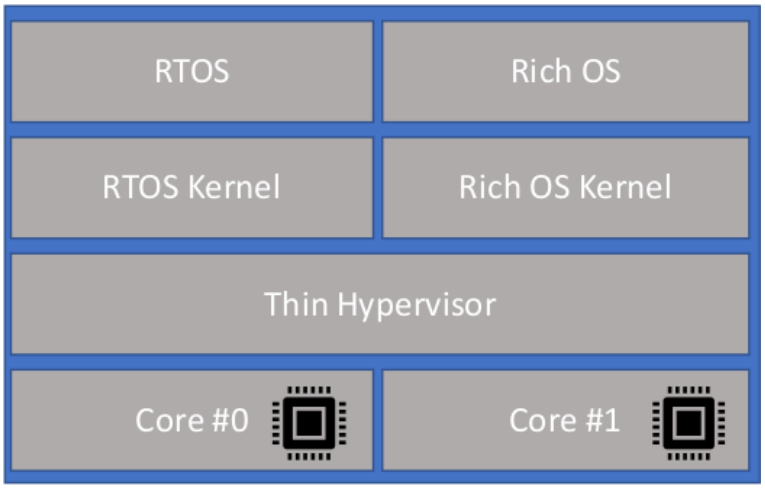


FIGURE 16 Embedded Partitioning

A partitioning example is Jailhouse. Jailhouse is a microvisor created by SiemensTM. It uses partitioning to split the hardware into isolated compartments, or cells. These cells are dedicated to executing programs called inmates. Jailhouse does not emulate any device. Devices and processors are statically assigned to Jailhouse on creation. In Jailhouse, a single processor is assigned to perform the hard real-time tasks while the other processors are assigned to run Linux. Jailhouse is implemented as a Linux kernel driver and is a type-2 hypervisor.

4 SUMMARY OF ORIGINAL PAPERS

This chapter presents a summary of 8 papers. Each section starts with the research problem, followed by the techniques we chose to approach it, and ends with the results.

4.1 Reverse Engineering Protection in ARM

Raz Ben Yehuda and
Nezer Jacob Zaidneberg

4.1.1 Paper Details

Protection against reverse engineering in ARM by Raz Ben Yehuda and Nezer Jacob Zaidenberg was published in *International Journal of Information Security volume 19, pages 39–51(2020)*
https://doi.org/10.1007/s10207-019-00450-1

Code
https://github.com/raziebe/TEE

4.1.2 Research Question

In the area of Anti-Reverse Engineering, code obfuscation is often used. However, experience from game consoles and computer games indicates that obfuscation eventually gets broken. To address this problem, we would like to prevent the reverse engineering of software by encrypting the software code before deployment, and deploying only the encrypted software, all under the following assumptions:

1. The encryption function we use is safe and cannot be broken. We use AES. However, we make no assumption about the encryption.

2. The CPU itself is sufficiently complex to prevent the attacker from "looking" inside the CPU.
3. We assume the CPU works according to the specifications, and there are no hidden modes allowing the internal CPU structures to be read.

4.1.3 Technique

Under the above assumptions, we showed that the decrypted software will not be available to the normal operating system, and protection for the software and the decryption keys will be undertaken by a hypervisor. This proposed system is composed of two phases:

Static Encryption
We choose which functions we wish to encrypt. Then we use TrulyProtect's Averbuch et al. (2013) instrumentation tool to encrypt the chosen functions on the binary copy of the program.

Runtime execution
The program runs as-is. Each time the encrypted function is accessed by the processor, the processor drops to HYP mode, decrypts the function, and executes from the exact line of code it entered the hypervisor. Any time the function performs a system call or tries to access memory that is not mapped to the hypervisor, the function is written onto with the escape sequence (pad-code), and then it shifts back to the normal world and resumes execution.

The main obstacle in this solution is performance. Repeatedly entering and exiting the hypervisor has a performance penalty. To lessen this penalty, the first time the hypervisor enters the encrypted code, it caches the decrypted code into a temporary protected buffer, and from then on the buffer is copied onto the pad-code each time the hypervisor is entered for execution. On exit, the function's decrypted instructions are flushed from the L1 instruction cache. In addition to caching the decrypted data, the hypervisor maps other parts of the process's address space into the hypervisor in real-time to reduce exits from HYP mode. Reducing the exits reduces the cache flashes and the copying of data. In addition, due the nature of the cache accesses, and since interrupts are disabled while in the hypervisor, predictability of execution is also evaluated.
An interesting approach to protect applications in a compromised operating system is overshadowing Chen et al. (2008). Overshadowing is a technology that creates two distinct views of the memory, one is the application view, and the other is the kernel view. Overshadowing utilizes virtualization.
One core concept of this technology is the cloaked application. A cloaked application is a program that is kept in its encrypted form in the disk, and when it executes, it is being decrypted by the hypervisor's VMM. A cloaked application can access the code and the data, while any other entity in the guest operating system can't. When the system tries to access a certain page, it faults into the

VMM, the VMM encrypts the page atomically and maps it to the system view. When the application tries to access the same page, it faults into the VMM, the VMM decrypts the page and remaps it to the application view. At this point, any program that accesses the page faults also into the VMM, and the VMM maps the encrypted page to the program view.
This technique resembles this paper technology in some ways. Both technologies offer different views of a page in the same system, and do not require recompilation of the program. However, the technologies differ in the following aspects:

1. This paper technique does not require a special loader as the overshadowing technology requires.
2. This paper technique is not bounded to page granularity, and any code or data address can be decrypted.
3. This paper technique is not vulnerable to DMA attacks, because it utilizes the processor cache while overshadowing does not.
4. This paper technique does not need to trap system calls, while overshadowing does. There is no need to marshal arguments.
5. This paper technique does require a SHIM layer. We glue only the encrypted code to binary.
6. Overshadowing travels between cores, while in this paper technology, we need to lock or re-decrypt in case of migration between cores.

4.1.4 Results

Our evaluations were performed on HiKey board over Debian Linux. We evaluated the IPA overhead, stack access penalty, VM enter/exit penalty, allocating and freeing memory, File Open , File Close, File Read, File write, and the time duration predictability of the encrypted function. The measurements showed a mild overhead in CPU usage. We also showed that our thin hypervisor is valid for I/O intensive workloads. Obviously, we can also assume that the encrypted sections are large, and as such the constant padding and decrypting lasts longer as the function size increases. We achieved this by minimizing the amount of context switches from the hypervisor to EL0.

4.1.5 Future work

The system is operational (So there is not much to do). We believe that performance improvement can be achieved using buffered execution as in Kiperberg et al. (2019b). However, since ARM architecture allows cheaper context switches than Intel architecture, the gains will be less significant and offset by the loss of cache.

4.2 The hyplet

Raz Ben Yehuda and
Nezer Zaindenberg

4.2.1 Paper Details

A poster, "Hyplets - Multi Exception Level Kernel towards Linux RTOS" by Raz Ben Yehuda and Nezer Zaidenberg was published in *SYSTOR '18: Proceedings of the 11th ACM International Systems and Storage Conference. June 2018*
https://dl.acm.org/doi/10.1145/3211890.3211917
A conference paper "The hyplet - Joining a Program and a Nanovisor for real-time and Performance" by Raz Ben Yehuda and Nezer Zaidenberg was published in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). 2020*
https://ieeexplore.ieee.org/document/9203743

Code
https://github.com/raziebe/rasp-hyplet

4.2.2 Research Question

In Linux and other operating systems, when an interrupt reaches the processor, it triggers a path of code that serves the interrupt. Furthermore, in some cases, the interrupt ends up waking a pending process. The challenge of reducing the latency between the interrupt event to the program is one research problem this paper approaches. A second research problem this paper tackles is reducing the RPC (Remote Procedure Call) latency between two processes in the Linux operating system. The RPC mechanism handles the parsing and handling of parameters and the returned values. Here, we only consider the local case and do not consider network communication. IPC in real-time systems is considered a latency challenge. Thus, system developers refrain from using IPC in many cases. The solution many programmers use is to put most of the logic in a single process. This technique decreases the complexity but increases the program size and risks. In multicore computers, one reason for the latency penalty is because the receiver may not be running when the message is sent. Therefore, the processor needs to perform at least two context switches.

4.2.3 Technique

The hyplet is an implementation of sharing a hypervisor address space with a standard Linux program. It adds a hypervisor awareness to the Linux kernel and executes code in the EL2. It is a new innovative technique to reduce the latency of sending an event from some execution context to another. An execution context may be an interrupt in kernel space or a user space process. The hyplet is a

function mapped to the hypervisor address space and some process. We can say that the hyplet is dually mapped to hypervisor and the operating system. The hypervisor has no view of a process but only the functions mapped, therefore we refer to this state as an asymmetric view of a process. Because the functions are mapped to the hypervisor, the hypervisor may execute them without being subordinate to the operating system kernel heuristics. The hyplet is the technology we created to achieve this.

hypISR
As the interrupt reaches the processor, instead of executing the user program code in EL0 after the ISR (and sometimes after the bottom half), a special routine of the program is executed in HYP mode at the start of the kernel's ISR. The hyplet does not change the processor state when it is in interrupt mode; thus, once the hyplet is completed, the interrupt can be processed in the kernel as originally intended. The hyplet does not require any new threads and because the hyplet is an ISR, it can be triggered in high frequencies. As a result, we can have high-frequency user-space timers in small embedded devices. While some may argue that the hyplet should have been implemented as a virtual interrupt, we chose to use upcalls, which are similar to hyperupcalls Amit and Wei (2018). The reason is because many ARMv8-a platforms do not support VGIC (virtual Interrupt Controller). Raspberry PI3, for example, does not fully support VGIC (as a consequence, ARM-KVM does not run on a RaspberryPi 3). Interrupts are then routed to the hypervisor by upcalls from the kernel main-interrupt routine, and the nanovisor communicates with the guest through a hyperupcall.

hypRPC
HypRPCs are intended to reduce the RPC latency to the sub-microsecond on average by eliminating the context switch (in some way the hyplet is viewed as a temporary address-space extension to the sending program). Here, a serving process registers a routine as a hyplet RPC (hypRPC), and at this point any authorized requester may invoke this procedure on demand. Upon a serving program exit, the API immediately returns an error. If the function needs to be replaced in real-time, there is no need to notify the requesting program; instead, the function in the hypervisor only needs to be replaced.

4.2.4 Results

In Figure 7 we demonstrated that the hypISR has the least latency compared to other operating systems.
Figure 17 compares the hypRPC to seL4 and a to a reference call function (Ref)-complete round trip. Figure 17 depicts an hypRPC benchmark on Raspberry Pi3. In the hypRPC case, 99.96 % of the samples were below the 2 microseconds, and 100% below 5 microseconds. The deviation was probably due to the interrupt being late.

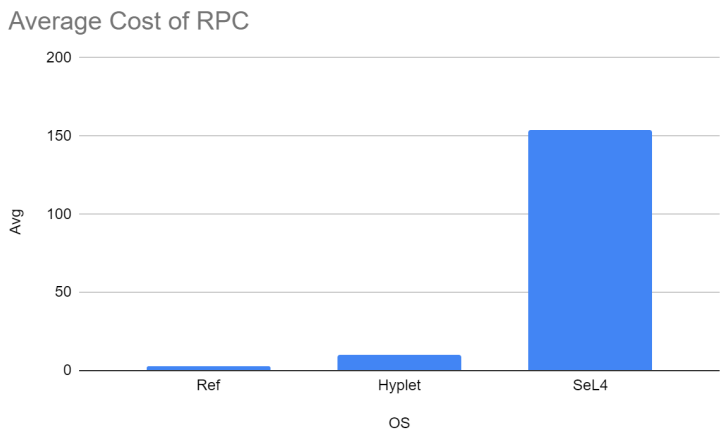


FIGURE 17 RPC durations

While the average hypRPC is 520 nanoseconds, in seL4 the average cost is 153 us.
To summarize, we showed that the hyplet is useful for real-time and performance.

4.2.5 Future work

Work on the hyplet is completed. We are working on a complete journal paper with more performance data. Furthermore, we are looking for more systems to implement hyplet based systems and demonstrate its benefits.

4.3 Memory Acquisition

Raz Ben Yehuda,
Erez Shlingbaum,
Shaked Tayouri,
Yuval Gershfeld and
Nezer Jacob Zaidenberg

4.3.1 Paper Details

A Journal paper "Hypervisor Memory acquisition for ARM" by Raz Ben Yehuda, Erez Shlingbaum, Shaked Tayouri, Yuval Gershfeld, Nezer Jacob Zaidenberg is accepted by Forensic Science. International: Digital Investigation. 2021. It is about to be published.

Code
https://github.com/raziebe/rasp-hyplet
https://github.com/raziebe/LiME

https://github.com/raziebe/volatility

4.3.2 Research Question

Volatility is an open-source (GPLv2) framework for analyzing memory Aljaedi et al. (2011). It is a forensics toolkit used to analyze memory snapshots. Thus, Volatility is often used to detect such hidden malware analysis. These images are acquired by various tools, one of which is LiME. LiME is a memory acquisition Linux driver, and our research improves the way it acquires memory images. We enhanced LiME's Sylve (2012) memory acquisition tool for Volatility Farmer and Venema (2009) memory forensics software. Since it is a driver, it is subordinate to the operating system heuristics, and therefore sometimes fails to produce coherent images. In addition, LiME consumes many CPU cycles and pressures the disk and/or network. We propose a hypervisor-based memory acquisition tool that fixes the above failures. Our implementation improves the creation of memory images. It reduces the processor's consumption, solves the in-coherence problem in the memory snapshots, and mitigates the pressure of the acquisition on the network and the disk.

Our system takes a precise, atomic memory image of an online system without stopping or freezing it. Furthermore, it is less suspect to acquisition errors as our technique is done outside the operating system, i.e. in the hypervisor. Therefore, the acquisition process is not vulnerable to stealth malware hiding.

4.3.3 Technique

o solve the memory images inconsistency, we wrap the GPOS, such as Linux or Android, with a minimal virtual machine. We record and copy the faulting page content before it is transmitted. This technique is also explained in the paper PVII. However, we extended it to ARM and Volatility. In virtual machines, when a guest accesses a page, the Memory Management Unit (MMU) traps it to a secondary page table managed by the hosting machine. This mechanism is referred to as a stage 2 fault. Thus, to achieve coherence, we set all stage 2 tables to read-only before the memory acquisition starts. This way, any write access to any page is trapped in the microvisor. The hypervisor uses an identity-mapping secondary-level page table. We then execute LiME to acquire the computer's RAM; at this point, any page that traps in the microvisor is copied, and the real page permissions are set to read-write. Therefore, each page can trap into the microvisor at most once.
LiME performs the acquisition linearly. However, linear acquisition has a flaw: the processor consumption is very high. To reduce the processor consumption, we modified LiME's main transmission routine to be less CPU intensive. In this approach, LiME does not scan the memory linearly. Instead, LiME removes pages from the pages pool as long as there are pages in it. If there are no pages, it linearly sends pages as before, but it sleeps for a few milliseconds in each transmission cycle. After each cycle, the pool is checked for accessed pages. Any page accessed

by LiME is verified as not re-transmitted. This is done by checking its permissions bits in the stage 2 table; If the page was transmitted then it is writeable.

4.3.4 Results

Our work demonstrated coherent images. We also show that our microvisor does not pose any visible load on the computer performance. Lastly, since our techniques relaxed the need to acquire memory fast, we were able to perform acquisition slowly without losing coherency and not stress the processor at all. This also makes it possible to send a complete memory image over a slow wireless network, i.e. this technology fits smartphones.

4.3.5 Future work

The work on forensics continues. We seek to merge the TrustZone attack paper and create Hyperleech Palutke et al. (2020) solution. Also, we are looking at better detection of attacks via machine learning and neural networks. Leon et al Leon et al. (2021) demonstrated that even without machine learning better detection of malware can be achieved.

4.4 The Offline Nanovisor

Raz Ben Yehuda and
Nezer Zaidenberg

4.4.1 Paper Details

The original work "The offline scheduler for embedded transportation systems" by Raz Ben Yehuda and Yair Wiseman was based on the author master thesis and published in *IEEE International Conference on Industrial Technology 2011 Mar 14 (pp. 449-454).* An extended version, "The offline scheduler for embedded vehicular systems" by Raz Ben Yehuda and Yair Wiseman was published in *International Journal of Vehicle Information and Communication Systems. 2013 Jan 1;3(1):44-57.* An extension based on the author recent work on ARM architecture and Hyplets is submitted.

Code
https://github.com/raziebe/rasp-hyplet

4.4.2 Research Question

Obtaining predictable latency while processing data is a challenging task. This is especially true in general-purpose operating systems. In GPOS, the program becomes less predictable, in many cases, due to cache and TLB misses Ekman

et al. (2002); Bennett and Audsley (2001). Software architects handle this unpredictability with various techniques. Such techniques include microkernels, microvisors and partitioning or real-time extensions to the GPOS. Other solutions include auxiliary processors, such as DSPs and even GPUs. Even current hard real-time operating systems do not provide high-resolution timers (for example 20Khz timers) that execute user space procedure in each tick. This is mainly because there is the cost of interrupt latency and context switch, which may reach a few microseconds. As an example, interrupt latency may reach 10 us, which is too high for a high-resolution timer. This work demonstrates a user-space timer in frequencies up to 20 kHz, with jitter of a few hundred nanoseconds.
Microcontrollers trap real-world events. These microcontrollers send interrupts to the GPOS processor to inform about upcoming events. Thus, the accuracy of the data also relies on the rate of the interrupts processed by the GPOS processor. In this context, therefore, we demonstrate the Offline Nanovisor as a technique to acquire data from external devices through high-speed data sampling. We construct test cases in which the physics change so fast that without a tight loop to access the data, it is not possible to observe these changes. For example, we show that we can measure the beginning and the return of an ultrasonic wave more accurately than a native program in a standard non-rtos Debian Linux.

4.4.3 Technology

This paper offers an alternative approach to the above for multi-processor machines. By virtually removing a processor from the operating system kernel, and running a single program in it, it is possible to achieve predictable latency in a GPOS. Removing a process from the GPOS scheduler and assigning it to a single task is referred to by Ben-Yehuda and Wiseman (2013) as the "offline scheduler". A difficulty with the offline scheduler design is that it can only run kernel code in what is referred to as an "offlet". We evolve the offline scheduler to process hyplets as introduced in Ben Yehuda and Zaidenberg (2018). Instead of running in kernel mode, the offlet executes in hypervisor mode. Hyplet technology is an excellent fit for the offline scheduler because both provide complementary real-time advantages. Furthermore, both run without interrupts, so it was easy to combine them. Therefore, we provide a hard real-time library in a Linux GPOS and a fast access to sensors that cannot generate interrupts, i.e. sensors that need to be polled. In the Offline Nanovisor, the kernel runs with interrupts disabled. Thus, the Offline Nanovisor guarantees latency as long as the program does not wait for another program or overflow the processor's L1 caches. An executing program never leaves the processor, and it is up to the programmer to yield the processor. As we show in this work, as long as the code and data remain in the processor's cache, real-time responsiveness is guaranteed. Furthermore, if the data size surpasses the processor's cache size, we can pre-fetch it (or part of it).

4.4.4 Results

We first start by demonstrating (Figure 18) the jitter when the operating systems are idle.

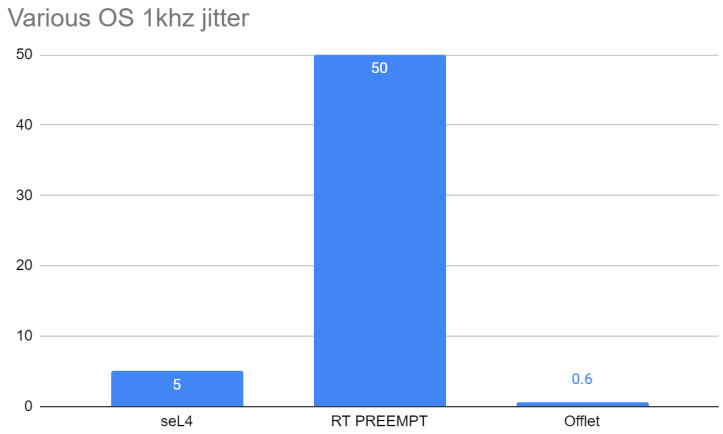


FIGURE 18 1Khz jitter (us) in a Raspberry Pi3

From Figure 18 we can see that moving to a 20Khz (50μs) timer resolution in Linux RT PREEMPT is not feasible in Raspberry Pi3. In seL4 the jitter reaches 10%, and the hyplet-offlet jitter is 0.6 μs, which is considerably less than seL4. Therefore, we continue to perform our tests in an offline hyplet with high frequencies.
Figure 19 depicts the results of the hyplet-offlet in various frequencies. The values were taken from an oscillator and not the system clock to remove any possible dependencies. To summarize, the hyplet-offlet can run user space timers in a standard Debian Linux with 600 nanoseconds' jitter at most.

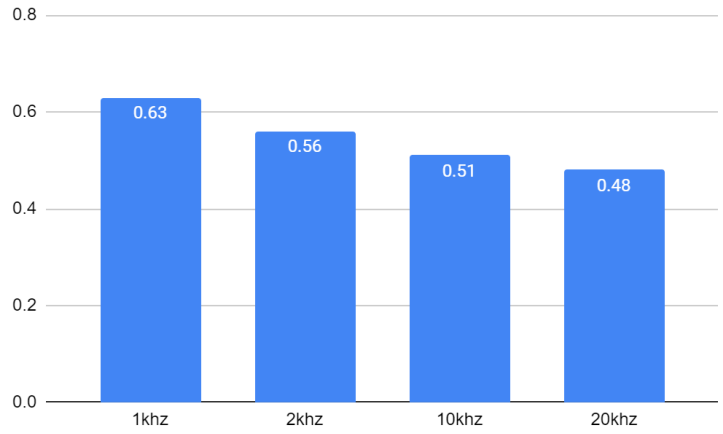


FIGURE 19 The hyplet-offlet in various frequencies

Another result our research shows is that physical phenomena can be measured by the Offline Nanovisor instead of expensive hardware. For example, we

show that we can measure the time delta between the time to trigger an ultrasonic beam until the moment the photo-resistor acknowledges (raised to 1) that the light has reached it. This is done by triggering a GPIO and reading another GPIO. Our research shows 1-microsecond standard deviation.

4.4.5 Future work

Our work continues and we utilize the Offline Nanovisor to transmit data over an ultrasonic medium Guri et al. (2014) and leak data from devices. Our Nanovisor can process radio signal in high accuracy to produce sensible data.

4.5 C-FLAT Linux

Raz Ben Yehuda,
Adam Aronov,
Or Ekstein and
Nezer Zaidenberg

4.5.1 Paper Details

The paper "NANOVISED C-FLAT Linux " is submitted.

Code
https://github.com/raziebe/hyplet-virt

4.5.2 Research Question

C-FLAT by Abera et al. Abera et al. (2016) is a technique for attestation of the control flow of an application running on an embedded device. C-FLAT is a dynamic analysis tool. It complements static attestation by capturing the program's run-time behavior and also verifies the exact sequence of executed instructions, including branches and function returns. It allows the verifier to trace the program's flow control to determine whether the application's flow was compromised. Combined with static attestation, C-FLAT can precisely attest embedded software execution.
Originally, C-FLAT design allows attestation for simple systems. Its original design does not support threads, processes or operating systems. Therefore, most industrial systems are too complex for C-FLAT. Such systems usually require a General Purpose Operating Systems, multiprocessing, multithreading, inheritance or function pointers et cetera. C-FLAT does not support any of these.
Furthermore, C-FLAT runs on top of TrustZone. TrustZone programming requires high expertise as well as access to the boot loader code. Such access is usually available only to the SoM (System On a Module) vendor. Here, we provide a similar but more straightforward approach through the use of our dedi-

cated Nanovisor Ben Yehuda and Zaidenberg (2020).
The paper presents the implementation of control flow attestation (C-FLAT) for Linux. We extended the design and implementation of C-FLAT through the use of a type-2 Nanovisor in the Linux operating system.

Threat model
An attacker may exploit a vulnerability to change a program's flow control. In this paper, a CFI (control flow integrity) attack refers to an attack on a native ELF (Executable Linkable Format) binary. For DOS attacks, we assume that an HTTP request passes the firewall if it exists. Our contribution provides software-only CFI for ARM in a GPOS and performing path monitoring in TEE. It is a context-sensitive CFI and is available for 32bit applications. It is helpful on Android Phones. Our Nanovisor can easily be ported to TrustZone.
C-FLAT is able to track suspicious paths; therefore it is useful to detect DOS attacks. The challenge here is two-fold. First is to adapt C-FLAT for Linux as well as minimize the performance penalty.
The second challenge of this work is to adapt C-FLAT for complex applications in a GPOS. We demonstrate how our system can detect real exploits such as SlowLoris, that affect production systems, and handle a real test case (CVE-2019-9210).

4.5.3 Technique

We first explain in brief the original C-FLAT, then we explain our innovation.

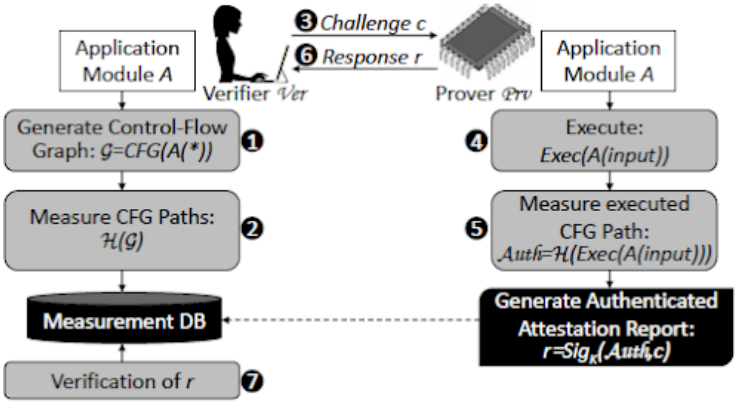


FIGURE 20 C-FLAT ©abera

In the first step (1) Ver performs offline processing and generates a control-flow-graph. It also measures each possible control-flow path through a measure function H, and stores the results in a measurement database (2). In the original C-FLAT paper, Ver stores and generates the CFG results in Prv, because Prv is a low-resources computer. In C-FLAT Linux the CFG is stored in Ver itself. This needs to be done only once per program.
In step (3), Ver challenges Prv. Then, Prv initiates execution of the program (4),

while a dedicated and trusted measurement engine computes a cumulative authenticator Auth of the control-flow path (5). In the original C-FLAT, this computation is performed in TrustZone, while here it is performed in a hypervisor. Lastly, Prv generates the attestation report r = SigK(Auth, c), computed as a digital signature over the challenge c and Auth using a key K known only to the measurement engine. Finally, Prv sends r to Ver (6) for validation.
As we aimed to mitigate the traps penalty, we needed to diminish the amount of Nanovisor traps (calls). Thus, we trigger the trap code off and on, so that it will not run all the time. To do that, we intervene in the protected program execution code while it executes, and replace the trap opcode (the BKPT instruction) with the NOP (no operation) opcode. A NOP opcode depends on the compiler output, and therefore to modify the opcode in real-time, it is required to know in advance: the size of the BKPT opcode (16 bit or 32 bit), the position of the opcode in the program's address space, and to identify that a protected program runs (or sleeps) without the need to wait for traps. We record the control flow path and send continuous sub-sequences to an attestation server. Furthermore, as a result of using Linux, the attestation server may execute locally. Local execution reduces the risk of transmitting the data over the network and allows for better performance.

4.5.4 Results

We demonstrated on a real test case — CVE-2019-9210 — and showed how this technology can be used to detect vulnerabilities. This CVE was found in advpng, a utility that is part of AdvanceCOMP, a compression and deflation tool suite. This CVE is a memory overrun caused by erroneous png files. We instrumented most of the advpng utility and executed it on png files in various dimensions. We then executed it on the erroneous input image file. C-FLAT was able to detect the incorrect control flow path because the advpng crashed. We present the performance results of the good (non-faulty) inputs and for the erroneous inputs. We did not perform any non-protected runs because advpng is a command-line tool whose execution ends in a few seconds in many cases. The two tables (Figure 1 and 2) depict the overhead of C-FLAT Linux.

TABLE 1 Good Input (ms) for advpng

Test	Avg	Max	Min	Std dev
Native	1628	1688	1606	31.8
Protected	2655	2795	2625	52

TABLE 2 Erroneous Input (ms) for advpng

Test	Avg	Max	Min	Std dev
Native	21.7	23	21	0.67
Protected	113	119	111	3

It is evident that C-FLAT has a big overhead when full protection is used. We also tested C-FLAT in detecting DOS attack, known as SlowLoris. C-FLAT detected the anomaly in the code flow, and since the web server spent lots of IOs, the time spent in the C-FLAT system is negligible compared to network IO.
We summarize that under heavy IO programs C-FLAT Linux performs well.

4.5.5 Future Work

An interesting new approach to CFI would be to utilize ARM's CoreSight to perform the CFI. ARM's CoreSight is a set of tools used to trace software that runs on Arm-based SoCs. We believe it is possible to manipulate the CoreSight to perform CFI.

4.6 HyperWall

Michael Kiperberg,
Raz Ben Yehuda and
Nezer Jacob Zaidenberg

4.6.1 Paper Details

A paper, "HyperWall: A Hypervisor for Detection and Prevention of Malicious Communication" by Michael Kiperberg, Raz Ben Yehuda and Nezer Jacob Zaidenberg is published in *International Conference on Network and System Security. NSS. 2020*
https://link.springer.com/chapter/10.1007/978-3-030-65745-1_5
The paper was *Awarded best paper*

4.6.2 Research Question

Malicious Malware tend to communicate with its operators, thereby revealing itself to the operating system protection tools. Attacks on the kernel are typically via data attacks or code attacks. Code attack changes the executing code. Data attacks modify data structures to achieve privilege escalation, process hiding etc.

The size of the kernel and modules is a sufficient attack surface; therefore, vendors created new techniques to protect sensitive data from the kernel in isolated environments, such as ARM TrustZone and Intel's SGX, i.e. isolation in the hardware. This paper presents an attack through a kernel module, and a protection scheme using a hypervisor. The threat model is an attacker that has access to the kernel memory and may modify code and data by exploiting some vulnerability. The attack is simple: access the MMIO region of the network card, modifying the transmit control registers. The code to perform the attacks requires only memory read and memory writes. This is an intentionally simple flow, so that it can be used by code injectors.

4.6.3 Technique

Hyperwall is hypervisor software that protects network cards' control registers from being manipulated. Because both legitimate and malicious accesses to the network card registers are performed from the kernel-mode, they are difficult to distinguish from the viewpoint of the hypervisor. Therefore, the hypervisor determines the legitimacy of access based on the content of the transmitted packets. Only packets that were previously transmitted by user-mode applications through sendto, sendmsg and sendmmsg are considered legitimate. This is done by intercepting various user-space socket operations of transmission, performing a hash computation of the data and storing it in the safe place (which is not accessible to the kernel). As the kernel driver performs the MMIO access to transmit registers, a VM-exit takes place and the hypervisor performs the hash calculation on the data, and then seeks it in the hashes tree. If the hash is not found, the descriptor is nullified so that it will not be transmitted.

4.6.4 Results

The test was performed in an Intel i217 network card. The hypervisor used was type-1. The processor was Intel Core-i5-7500, Ubunto 64bit. In the LMbench microbenchmark, the overhead peaks to 160% in sockets I/O operations. Figure 3 depicts the LM benchmarks.

TABLE 3 Hyperwall LMbench

Test	Vanila Linux	Hyperwall	Overhead in %
syscall	1.7996	1.8844	5
read	0.4714	0.5892	25
write	0.434	0.5631	30
fstat	0.4724	0.5838	24
open/close	1.5462	2.4174	56
select (10 fds)	0.5395	0.7116	32
select (100 fds)	5.2287	9.3992	80
fork+exit	66.8446	80.216	20
fork+execve	210.5385	249.1556	18
fork+/bin/sh	541.8947	1336	147
sigaction	0.4423	0.5148	16
Signal	1.002	1.6766	67
Protetcion	0.823	1.142	39
Page	0.1495	0.3148	111
Unix	5.2694	7.8473	49
TCP	8.1917	18.4216	125
UDP	6.2157	16.4073	164

4.6.5 Future work

Work on hyperwall can proceed to include CFI based filtering. A CFI based filtering can provide more robust protection for an application can be provided. In the current solution, the operating system can read and write from application buffers using standard system calls (e.g. read(2) and write(2)) if CFI based protection is enabled, buffers can only be read or written if a correct CFI path is saved.

4.7 Hypervisor Memory Introspection and Hypervisor based Malware Honeypot

Nezer Zaidenberg,
Michael Kiperberg,
Raz Ben Yehuda,
Asaf Alagawi,
Roee Leon and
Amit Resh

4.7.1 Paper Details

A Book Chapter. "Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot" by Nezer Zaidenberg, Michael Kiperberg, Raz Ben Yehuda, Asaf Algawi, Roee Leon and Amit Resh was published in *Communications in Computer and Information Science book series (CCIS, volume 1221). 2020.*
The chapter is available at
https://link.springer.com/chapter/10.1007%2F978-3-030-49443-8_15

Code
https://github.com/raziebe/rasp-hyplet

4.7.2 Research Question

The paper deals in memory acquisition in x86. However, triggering memory collection while malware starts is difficult because it is not easy to identify at which point in time the malware starts. Some malware tries to utilize the machine to gain access to other machines, usually by trying to access the network. Therefore, researchers suggested using honeypots as network cards. However, viruses such as StuxNet Langner (2011) attack may try to access storage devices and install replicates in them. This work suggests a honeypot that differs from prior ones as we use the Nanovisor to create honeypot devices within the machine instead of the network. This innovation allows us to trigger memory collection just as the malware starts operating.

4.7.3 Technique

We created a virtual disk. By using the same technique as in the memory acquisition research, the disk's memory is protected by the hypervisor. A process that may access the virtual disk is white listed, and is not reported. In our implementation, the virtual disk is a RAM drive. Its memory footprint is small and filled with zeroes. There is no need to use a big footprint because we do not write anything to the disk. We utilized the hyplet for this purpose, and controlled the white lists and monitors in a regular Linux process with a hyplet. The lists are controlled in the hypervisor and mapped to the hypervisor and cannot be accessed in EL0. In addition, the virtual disk differs from the regular memory acquisition technique in that the access rights do not change to read-write when the device is accessed. The fact that the memory is protected in the hypervisor means that even a kernel virus will fail to manipulate it.

4.7.4 Results

The disk reported any malware trying to access it. To evaluate the overhead of using IPA we used RAMspeed. In Table 4 we attempted to simulate more closely real-world computing load. A, B and C are locations in the memory. In SCALE m is a constant.

TABLE 4 SLAT Real world load GB/s

Test	COPY A = B	SCALE A = $m \cdot B$	ADD A + B = C
Single Stage	2.76	1.52	2.60
Dual Stage	2.74	1.52	2.53

Table 4 shows that IPA overhead is negligible if any. To summarize, using a thin hypervisor for any purpose, does not pose any overhead for the system.

4.7.5 Future work

The author other works on memory forensics is focused on ARM platform. This work is an exception (as it is based on intel architecture). Thus, the author does not expect future work in this area unless motivated by the other authors.

4.8 Attacking TrustZone

Ron Stajnrod,
Raz Ben Yehuda and
Nezer Jacob Zaidenberg

4.8.1 Paper Details

The paper "Attacking TrustZone" had been submitted.

Code
https://github.com/ronst22/dma_repo

4.8.2 Research Question

ARM's TrustZone is an excellent way to implement security mechanisms across IoT-embedded devices. Nevertheless, it is still prone to bad hardware and software implementations; thus, the hardware of different companies like Google, Samsung, Huawei, etc. might still be affected by severe vulnerabilities that compromise the entire security suite Shen (2015); Chen et al. (2017); Makkaveev (2020); Guilbon (2020).
Some ARM modules lack AMBA AXI ARM. (2020) support, and do not follow the APB specifications (Figure 5), this may lead to insecure memory separation between the normal and secure worlds. We present a Direct Memory Access (DMA) attack Kupfer et al. (2018) on ARM TrustZone Trusted Applications (TA) running the OP-TEE. The DMA allows an attacker to execute arbitrary code in the secure

world or read arbitrary data from the secure world into the Rich OS. Our attack is a control-flow attack Davi et al. (2014) Zhang and Sekar (2013) on the OP-TEE kernel. This research is a walk-through tutorial for a DMA attack.

4.8.3 Technique

We attacked an OP-TEE (Open Portable Trusted Execution Environment) OS. OP-TEE is designed as a companion to a non-secure Linux kernel running on ARM Cortex-A cores using the TrustZone technology.
Each TA (Trusted Application) is encrypted by a private key. Under the OP-TEE framework, the decryption takes place in the TrustZone. Thus, the program in its decrypted form is only visible in the Secured RAM and the processor's EL3 cache. It is, therefore, sensible to attack in the decryption area (the OP-TEE kernel).
Direct Memory Access (DMA) allows I/O devices to access the memory. DMA capable devices incorporate DMA engines that enable them to initiate DMA transactions without the coordination of a central DMA controller. The attack goal is to exploit TrustZone. We escalate privileges by reading data from the Secure World. Through this attack, we inject code to the Monitor in EL3, thus executing malicious programs in the secure world operating system (the secure world kernel). This offers us to bypass any validation of the secure operating system, patch the EL3 kernel and perform arbitrary code execution (ACE).
The attack is based on the *Write What Where* vulnerability achieved using DMA transactions. We use this vulnerability to show that we can gain access to execute arbitrary code in the OP-TEE_OS, thereby bypassing OP-TEE OS trusted application signature validation and gaining control of every trusted application in the system. Our approach is to change the return values of key functions without changing the stack. This technique impedes CFI tools such as gcc stack guard Cowan et al. (1998) or Clang Moreira et al. (2017) kFCI to detect our attack.
Our DMA attack is initiated by the CPU, through a Linux kernel driver. After reading memory pages from the RAM, we analyse the memory and compare it to ARM Trusted Firmware in order to locate similar functions. (Most of the TrustZone software implementations are based on ARM Trusted Firmware, which makes reverse-engineering of the code simpler.) Moreover, there are some major vendors' secure OS (Trusted Execution Environment) in the market (QSEE, OP-TEE) and we compare our memory dump to the compiled versions of those; by doing so, we can find the functions that validate trusted application signatures. Because none of the widely used TEE OS use Address Space Layout Randomization (ASLR) Cook (2013), we can use the address from our memory dump to override trusted applications' signature validations with a DMA attack. After doing so, we can just replace any TA with our own malicious TA. Even though we do not know the correct signature private key, the TEE OS will succeed in validating our malicious TA. Our attack is concentrated on the monitor code that validates the TA signatures, size, and other properties of the TA.

The attack was performed on Raspberry Pi3 because Raspberry PI does not support physical bus secure world separation, and is widely used.

4.8.4 Results

Trusted applications consist of a signed ELF header, named from the UUID of the trusted application (set during compilation time) and the suffix .ta. The trusted applications binaries are on the REE file-system and, thus, can be overwritten; However, the OP-TEE OS never executes a replaced TA when the signature does not match. Faking a matching signature requires finding a private key of *2048-bit* that matches the public key; therefore, only the owner of the key would be able to replace those applications.
In our case, we compiled a new TA with the same UUID as the original one and put it in the file-system location. By executing our malicious TA, we gained the ability to manipulate ARM TrustZone to execute invalid signed binaries. For example we compiled a fake TA (given in the examples of the OP-TEE suite) that encrypts data with our malicious key. Thus, every time the user uses this TA to perform AES, the data will not be truly encrypted with a secret key.

4.8.5 Future Work

We will continue the PMU work and utilize it to install a hypervisor on smartphones (such as Qualcomm SoMs) that do not provide access to EL2. We will then add C-FLAT or hyperwall on these phones.

5 CONCLUSIONS

We can divide our conclusions into two parts: The first part is more general and deals with the principles we've learned regarding the structure of academic research. The second is that this dissertation is one step in a ladder of the ARM processor's research.
We learned that it is important to examine others' work, and look for ways to improve it, or in some cases, try to attack it, and thereby improve it. We also learned that combining hands with other fellow researchers, reading their work, and trying to apply their methods and techniques into our fields of research. C-FLAT and memory acquisition are good examples of these kinds of efforts.
We also learned to leverage criticism to improve our research. In many cases, the way the data is presented is important to the reviewer. For example, we usually presented raw data in histograms, but the reviewer asked us to present it also in a graph. There were times that criticism required that we extend our research. For example, in the C-FLAT research, a reviewer asked us to add a benchmark.
Regarding the content of this dissertation, we learned that it is possible to change the original use of the processor's functionality to other uses. We also learned that applying techniques from the x86 architecture to the ARM architecture creates new ideas.
From our TrustZone attack, we feel it is also interesting to continue researching the new ARM features, such as the PAM ARMv8.3-PAuth, and check what this extension does not cover, which, in this case, large tables. We can create an attack, and then provide a solution.
We also conclude that the hyperwall is an interesting security extension that can be useful for smartphones. The hyperwall can protect network cards, mainly wireless, and because wireless cards do not generate heavy traffic the hyperwall will not overload the device. Memory acquisition is also useful for smartphones and may become a new feature to the Android OS.

YHTEENVETO (SUMMARY IN FINNISH)

Tämä väitöskirja tutkii ARM-virtualisointia ja TrustZone- laajennuksia. Laajennamme virtualisointia saadaksemme turvallisuutta ja reaaliaikaisuutta sekä tutkimme tapoja rikkoa TrustZone. ARM on lyhenne sanoista Advanced RISC (Reduced Instruction set Computing) Kone. RISC-prosessorit vaativat vähemmän transistoreita kuin CISC (Complex Instruction Set Computing). RISC vähentää kustannuksia, vähentää virrankulutusta, ja tuottaa vähemmän lämpöä, joten se sopii hyvin pienitehoisiin laitteisiin. Siksi se ei ole yllättävää, että esim. älypuhelimissa on ARM käytössä. ARM-markkinat hallitsevat tänään IoT (esineiden Internet) -markkinoita ja se on hyvin yleinen ajoneuvo- ja robotiikkamarkkinoilla. Sen osuus pilvipalveluissa kasvaa hitaasti. ARM:n käyttö Applen IPhonessa ja Googlen Androidissa on lisännyt sen leviämistä nopeasti. Älypuhelinteollisuus ja sen kanssa ARM-ytimet yleistyivät nopeasti 2010 luvun kieppeillä, mutta sen myötä samoin tekivät myös turvallisuusriskit. Valmistajien oli pakko investoida paljon enemmän resursseja puhelimien turvaamiseen. ARM-arkkitehtuuri alkaa versiosta ARMv1 ja ulottuu ARMv8: een. Näillä arkkitehtuureilla on erilaisia malleja, kuten Cortex-perhemalli. Mallit vaihtelevat nopeudeltaan ja eri arkkitehtuurit eroavat ominaisuuksiltaan. Ennen älypuhelimen ilmestymistä ARM:ia käytettiin pääasiassa sulautetuissa laitteissa. Sulautetun teollisuuden merkittävät mutta pienet käyttöjärjestelmät ovat FreeRTOS, VxWorks, Zephyr jne. Symbiania, yhtä merkittävää käyttöjärjestelmää käytettiin Nokia-puhelimissa, toimi ARM: lla. Sulautetut laitteet ovat taipuvaisia palvelemaan yhtä tarkoitusta, eivätkä siksi vaadi GPOS:aa (General Operating System). Kuitenkin Androidin ja IOSin yleistyttyä GPOS:n merkitys sulautetuissa laitteissa on hiipunut. On aivan ymmärrettävää, että Android ja iPhone OS (IoS) ovat yleisimpiä ARM-ytimiä käyttäviä käyttöjärjestelmiä tänä päivänä. Nämä käyttöjärjestelmät laajensivat ARM:in käyttöä. Sitä ennenn oli olemassa esimerkiksi Java ja Python, kehykset, kuten QT (monialustainen alusta), Unity (pelialusta), ja monet muut. Muita tunnetuja ARM:ia käyttäviä käyttöjärjestelmiä ovat Windows for ARM, Linux ja FreeBSD. Tämä väitöskirja osoittaa luotettavan suorituksen ARM:lla hypervisorin kautta. Uusi suorituskonteksti esitellään, jota kutsumme hypletiksi. Hypletti on tekniikka jossa tavallisen Linux-prosessin toiminto suoritetaan hypervisorissa. Osoitamme myös uuden tekniikan yhtenäisten muistikuvien (RAM) hankkimiseksi kääyttämällä ARM-hypervisoria, samalla kun säästetään virtaa ja vähennetään lämpöä. Reaaliaikaisella alueella esittelemme offline-nanovisorin. Offline-nanovisor laajentaa hyplettiä suorittamaan käyttäjätilakoodin korkeilla taajuuksilla, esim. 20KHz yleisessä Linuxissa. CFI: n alueella C-FLAT on tekniikka puhtaasti sulautettuille laitteille. Osoitamme laajennoksen puhtaasta C-FLAT laitteesta Linuxiin. Hyperwall käsittelee hyökkääjän uhkaa, jolla on pääsy ytimen muistiin ja sillä voi muokkata koodia sekä tietoja hyödyntämällä käyttää joitain haavoittuvuuksia. Hyperwall suojaa verkkokortteja manipuloinnilta. Turvallisuuden alalla esittelemme myös tekniikan, joka luo hunajapotin virtuaalilevyjä käyttämällä. Lopuksi esitämme DMA-hyökkäyksen TrustZoneen.

REFERENCES

ARM. 2020. About the AXI TrustZone memory adapter. https://developer.arm.com/docs/dto0017/a/about-the-axi-trustzone-memory-adapter. (Accessed: 2020-04-15).

Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R. & Gene, T. 2016. C-flat: control-flow attestation for embedded systems software. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 743–754.

Algawi, A., Kiperberg, M., Leon, R., Resh, A. & Zaidenberg, N. 2019. Creating modern blue pills and red pills. In ECCWS 2019 18th European Conference on Cyber Warfare and Security. Academic Conferences and publishing limited, 6.

Aljaedi, A., Lindskog, D., Zavarsky, P., Ruhl, R. & Almari, F. 2011. Comparative analysis of volatile memory forensics: live response vs. memory imaging. In 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing. IEEE, 1253–1258.

Allen, S., Graupera, V. & Lundrigan, L. 2010. Pro smartphone cross-platform development: iPhone, blackberry, windows mobile and android development and distribution. Apress.

Amit, N. & Wei, M. 2018. The design and implementation of hyperupcalls. In 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 97–112.

Averbuch, A., Kiperberg, M. & Zaidenberg, N. J. 2013. Truly-protect: An efficient vm-based software protection. IEEE Systems Journal 7 (3), 455–466.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. & Warfield, A. 2003. Xen and the art of virtualization. In ACM SIGOPS operating systems review, Vol. 37. ACM, 164–177.

Baryshnikov, M. 2016. Jailhouse hypervisor. České vysoké učení technické v Praze. Vypočetní a informační centrum. B.S. thesis.

Ben-Yehuda, R. & Wiseman, Y. 2011. The offline scheduler for embedded transportation systems. In 2011 IEEE International Conference on Industrial Technology. IEEE, 449–454.

Ben-Yehuda, R. & Wiseman, Y. 2013. The offline scheduler for embedded vehicular systems. International Journal of Vehicle Information and Communication Systems 3 (1), 44–57.

Benbachir, A. & Dagenais, M. R. 2018. Hypertracing: Tracing through virtualization layers. IEEE Transactions on Cloud Computing.

Bennett, M. & Audsley, N. C. 2001. Predictable and efficient virtual addressing for safety-critical real-time systems. In Proceedings of the 13th Euromicro Conference on Real-Time Systems. IEEE, 183–190.

Ben Yehuda, R., Kevorkian, D., Zamir, G. L., Walter, M. Y. & Levy, L. 2019a. Virtual usb honeypot. In Proceedings of the 12th ACM International Conference on Systems and Storage. New York, NY, USA: Association for Computing Machinery. SYSTOR '19, 181. doi:10.1145/3319647.3325843. ⟨URL:https://doi.org/10.1145/3319647.3325843⟩.

Ben Yehuda, R., Leon, R. & Zaidenberg, N. 2019b. Arm security alternatives. In Proceedings of the European conference on information warfare and security. Academic Conferences International.

Ben Yehuda, R. & Zaidenberg, N. 2018. Hyplets-multi exception level kernel towards linux rtos. In Proceedings of the 11th ACM International Systems and Storage Conference, 116–117.

Ben Yehuda, R. & Zaidenberg, N. 2020. The hyplet - joining a program and a nanovisor for real-time and performance. In SPECTS, The 2020 International Symposium on Performance Evaluation of Computer and Telecommunication Systems Conference.

Ben Yehuda, R. & Zaidenberg, N. J. 2019. Protection against reverse engineering in arm. International Journal of Information Security, 1–13.

Ben Yehuda, R. & Zaidenberg, N. J. 2020. Protection against reverse engineering in arm. International Journal of Information Security 19 (1), 39–51.

Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A. & Heiser, G. 2011. Timing analysis of a protected operating system kernel. In Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd. IEEE, 339–348.

Blunden, B. 2012. The Rootkit arsenal: Escape and evasion in the dark corners of the system. Jones & Bartlett Publishers.

Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z. & Lai, T. H. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 142–157.

Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J. & Ports, D. R. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM SIGOPS Operating Systems Review 42 (2), 2–13.

Chen, Y., Zhang, Y., Wang, Z. & Wei, T. 2017. Downgrade attack on trustzone. In The 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2017).

Cook, K. 2013. Kernel address space layout randomization. Linux Security Summit.

Costan, V. & Devadas, S. 2016. Intel sgx explained. IACR Cryptology ePrint Archive 2016, 86.

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. & Hinton, H. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX Security Symposium, Vol. 98. San Antonio, TX, 63–78.

Dall, C. & Nieh, J. 2014a. Kvm/arm: The design and implementation of the linux arm hypervisor. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: ACM. ASPLOS '14, 333–348. doi:10.1145/2541940.2541946. ⟨URL:http://doi.acm.org/10.1145/2541940.2541946⟩.

Dall, C. & Nieh, J. 2014b. Kvm/arm: the design and implementation of the linux arm hypervisor. In ACM SIGARCH Computer Architecture News, Vol. 42. ACM, 333–348.

Dave, R., Mistry, N. R. & Dahiya, M. 2014. Volatile memory based forensic artifacts & analysis. Int J Res Appl Sci Eng Technol 2 (1), 120–124.

Davi, L., Koeberl, P. & Sadeghi, A.-R. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, 1–6.

Ekman, M., Dahgren, F. & Stenstrom, P. 2002. Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In Proceedings of the international symposium on Low power electronics and design. IEEE, 243–246.

Elphinstone, K. & Heiser, G. 2013. 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. ACM, 133–150.

Farmer, D. & Venema, W. 2009. Forensic discovery. Addison-Wesley Professional.

Gerum, P. 2004. Xenomai-implementing a rtos emulation framework on gnu/linux. White Paper, Xenomai, 1–12.

Guan, F., Peng, L., Perneel, L. & Timmerman, M. 2016. Open source freertos as a case study in real-time operating system evolution. Journal of Systems and Software 118, 19–35.

Guilbon, J. 2020. Attacking the ARM's TrustZone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html. (Accessed: 2020-04-16).

Guri, M., Kedma, G., Kachlon, A. & Elovici, Y. 2014. Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In 2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE). IEEE, 58–67.

Hall, S. P. & Anderson, E. 2009. Operating systems for mobile computing. Journal of Computing Sciences in Colleges 25 (2), 64–71.

Hambarde, P., Varma, R. & Jha, S. 2014. The survey of real time operating system: Rtos. In 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies. IEEE, 34–39.

Heiser, G. & Leslie, B. 2010. The okl4 microvisor: Convergence point of microkernels and hypervisors. In Proceedings of the first ACM Asia-Pacific workshop on systems. ACM, 19–24.

J.Flynn., M. 1995. Computer Architecture, PIPELINED AND PARALLEL PROCESSOR DESIGN.

Khen, E., Zaidenberg, N. J., Averbuch, A. & Fraimovitch, E. 2013. Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation. In 2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). IEEE, 78–85.

Kim, E. & Shin, D. 2018. Separation of kernel space and user space in zephyr kernel. IEMEK Journal of Embedded Systems and Applications 13 (4), 187–194.

Kiperberg, M., Ben Yehuda, R. & Zaidenberg, N. J. 2020. Hyperwall: A hypervisor for detection and prevention of malicious communication. In International Conference on Network and System Security. Springer, 79–93.

Kiperberg, M., Leon, R., Resh, A., Algawi, A. & Zaidenberg, N. 2019a. Hypervisor-assisted atomic memory acquisition in modern systems. In International Conference on Information Systems Security and Privacy. SCITEPRESS Science and Technology Publications.

Kiperberg, M., Leon, R., Resh, A., Algawi, A. & Zaidenberg, N. J. 2019b. Hypervisor-based protection of code. IEEE Transactions on Information Forensics and Security 14 (8), 2203–2216.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. & Winwood", S. 2008. sel4: formal verification of an os kernel. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: ACM. ASPLOS XIII, 168–178. doi:10.1145/1346281.1346303. ⟨URL:http://doi.acm.org/10.1145/1346281.1346303⟩.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T. et al. 2019. Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 1–19.

Kupfer, G., Tsafrir, D. & Amit, N. 2018. IOMMU-resistant DMA attacks. Computer Science Department, Technion. Ph. D. Thesis.

Kuz, I., Liu, Y., Gorton, I. & Heiser, G. 2007. Camkes: A component model for secure microkernel-based embedded systems. Journal of Systems and Software 80 (5), 687–699.

Lackorzynski, A., Weinhold, C. & Härtig, H. 2016. Combining predictable execution with full-featured commodity systems. In Proceedings of OSPERT2016, the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications OSPERT 2016, 31–36.

Langner, R. 2011. Stuxnet: Dissecting a cyberwarfare weapon. IEEE Security & Privacy 9 (3), 49–51.

Leon, R. S., Kiperberg, M., Leon Zabag, A. A. & Zaidenberg, N. J. 2021. Hypervisor-assisted dynamic malware analysis. cybersecurity 4 (19).

Liedtke, J., Elphinstone, K., Schonberg, S., Hartig, H., Heiser, G., Islam, N. & Jaeger, T. 1997. Achieved ipc performance (still the foundation for extensibility). In Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133). IEEE, 28–31.

Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E. & Asokan, N. 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In 28th {USENIX} Security Symposium ({USENIX} Security 19), 177–194.

Linaro 2020. Open Portable Trusted Execution Environment. http://www.op-tee.org/. (Accessed: 2020-09-30).

Makkaveev, S. 2020. The Road to Qualcomm TrustZone Apps Fuzzing. https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/. (Accessed: 2020-04-16).

Mantegazza, P., Dozio, E. & Papacharalambous, S. 2000. Rtai: Real time application interface. Linux Journal 2000 (72es), 10.

Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J. & Zwaenepoel, W. 2005. Diagnosing performance overheads in the xen virtual machine environment. In Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. ACM, 13–23.

Moreira, J., Rigo, S., Polychronakis, M. & Kemerlis, V. P. 2017. Drop the rop fine-grained control-flow integrity for the linux kernel. Black Hat Asia.

Müller, C., Felber, P., Cachin, C., Schiavoni, V. & Brandenburger, M. 2019. Execution of Smart Contracts with ARM TrustZone.

Ning, Z. & Zhang, F. 2019. Understanding the security of arm debugging features. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 602–619.

Palutke, R., Ruderich, S., Wild, M. & Freiling, F. 2020. Hyperleech: Stealthy system virtualization with minimal target impact through dma-based hypervisor injection. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020), 165–179.

Patel, A., Daftedar, M., Shalan, M. & El-Kharashi, M. W. 2015. Embedded hypervisor xvisor: A comparative analysis. In 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 682–691.

Payne, B. D. 2012. Simplifying virtual machine introspection using libvmi. Sandia report, 43–44.

Penneman, N., Kudinskas, D., Rawsthorne, A., De Sutter, B. & De Bosschere, K. 2013. Formal virtualization requirements for the arm architecture. J. Syst. Archit. 59 (3), 144–154. doi:10.1016/j.sysarc.2013.02.003. ⟨URL:http://dx.doi.org/10.1016/j.sysarc.2013.02.003⟩.

Popek, G. J. & Goldberg, R. P. 1974. Formal requirements for virtualizable third generation architectures. Communications of the ACM 17 (7), 412–421.

Qi, Z., Xiang, C., Ma, R., Li, J., Guan, H. & Wei, D. S. 2016. Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. IEEE Transactions on Cloud Computing 5 (3), 443–456.

Resh, A., Kiperberg, M., Leon, R. & Zaidenberg, N. 2017. System for executing encrypted native programs. International Journal of Digital Content Technology and its Applications 11.

Rostedt, S. & Hart, D. V. 2007. Internals of the rt patch. In Proceedings of the Linux symposium, Vol. 2, 161–172.

Russel, R. 2007. lguest: Implementing the little linux hypervisor. OLS 7, 173–178.

Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J. & Klein, G. 2011. sel4 enforces integrity. In International Conference on Interactive Theorem Proving. Springer, 325–340.

Shen, D. 2015. Exploiting trustzone on android. In Black Hat USA.

Spisak, M. 2016. Hardware-assisted rootkits: Abusing performance counters on the {ARM} and x86 architectures. In 10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16).

Stüttgen, J. & Cohen, M. 2013. Anti-forensic resilient memory acquisition. Digital investigation 10, S105–S115.

Sylve, J. 2012. Android mind reading: Memory acquisition and analysis with dmd and volatility. In Shmoocon 2012.

Waldrop, M. M. 2016. The chips are down for moore's law. Nature News 530 (7589), 144.

Winter, J. 2008. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In Proceedings of the 3rd ACM workshop on Scalable trusted computing. ACM, 21–30.

Zaidenberg, N. J., Ben Yehuda, R. & Leon, R. S. 2020. Arm hypervisor and trustzone alternatives. In Encyclopedia of Criminal Activities and the Deep Web. IGI Global, 1150–1162.

Zaidenberg, N. J., Kiperberg, M., Ben Yehuda, R., Leon, R., Algawi, A. & Resh, A. 2019. Hypervisor memory introspection and hypervisor based malware honeypot. In International Conference on Information Systems Security and Privacy. Springer, 317–334.

Zaidenberg, N. J. 2018. Hardware rooted security in industry 4.0 systems. Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures 51, 135.

Zaidenberg, N. J. & Khen, E. 2015. Detecting kernel vulnerabilities during the development phase. In 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing. IEEE, 224–230.

Zhang, M. & Sekar, R. 2013. Control flow integrity for {COTS} binaries. In Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), 337–352.

documentation arm 2021a. ARM. https://developer.arm.com/documentation/102142/0100/Virtualization-Host-Extensions. (Accessed: 2021-08-11).

documentation arm 2021b. DAIF. https://developer.arm.com/documentation/100933/0100/Processor-state-i-exception-handling. (Accessed: 2021-08-11).

ORIGINAL PAPERS

PI

THE OFFLINE SCHEDULER FOR EMBEDDED TRANSPORTATION SYSTEMS.

by

Raz Ben Yehuda, Yair Wiseman

Proceedings of Industrial Technology (ICIT), IEEE International Conference on.
2011

The Offline Scheduler for Embedded Transportation Systems

Raz Ben-Yehuda
BitBand LTD
Netanya, Israel
e-mail: razb@bitband.com

Yair Wiseman
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel
e-mail: wiseman@cs.biu.ac.il

***Abstract*—Nowadays, various transportation means use Linux as the operating system for their embedded control systems; however Linux uses all the processes in the hardware equally. This motivates some designer to develop an entire new Operating System for the vehicle; where a modification of Linux can produce the same and even a better product. This paper explains how this modification was designed and implemented in a commercial transportation company.**

I. Introduction

Over the last years, Linux has been becoming the Operating system of choice for many transportation systems. The Linux adaption incorporates a wide range of transportation means from small cars [1] to long trains [2,3]. Also, new embedded systems of airplanes employ Linux [4] and some of them even a Linux distributed system [5].

The adaptation of Linux has many advantages; however, Linux like other current operating systems has no easy way to assign a processor to do an individual service using an undisturbed, accurate and fast way and still being a part of the operating system address space as long as the processor is an active part of an operating system. We suggest a technique of dedicating one processor of a running machine to a specific task.

Amdahl's Law asserts that a linear speed-up is not likely to be practicable [6,7], so this can motivate in some cases sparing a processor for certain tasks. That is to say we can use the Linux kernel ability to virtually hot plug an (SMT) processor [8]; instead of letting the processor mark time in endless "halts", assign the processor a service. The offline scheduler treats a processor as a device with computation ability and this is why the processor can be offloaded.

II. Related Works

A. CPU Sets

Current common technologies for service oriented systems are Linux CPU sets [9,10] and Solaris Resource Pool association [11,12]. Both of these technologies refer to assigning tasks to a set of processors, probably on the same memory node (in the NUMA case). CPU sets are actually a constraint on a (user space/kernel space) task to use resources available only in its set. CPU sets are used mainly in big systems, appliances, adapted to the client needs, where performance is an important issue. Assigning tasks to a CPU set is done using simple interfaces for maintaining memory policies and there is a very small overhead for the programmer. CPU sets was not designed to run Real-Time tasks and do not intent to reduce the interrupts over-head. This is unlike the offline scheduler who is a Real-Time scheduler. The offline scheduler may act as an alternate to softirqs and ISR. It is serialized, very accurate and provides a controlled environment to the service. The offline scheduler also protects the operating system in some cases. Therefore, Linux CPU sets and Solaris Resource Pool association serve different purposes than the offline scheduler.

B. INtime

INtime RTOS [13] for Microsoft windows is an extension for Microsoft Windows running on any platform that utilizes a standard Microsoft HAL: uniprocessor or multiprocessor. INtime RTOS has many components similar to the offline schdeuler. INtime and the offline scheduler share this features:

- INtime is separated from Microsoft windows and offsched is separated from Linux; however both of the pairs share the same address space.
- Both INtime and offsched scheduler can allocate processor cores for exclusive use of INtime/offsched scheduler.
- Both INtime and offsched scheduler have high precision system timer of 50μs.

In spite of these, there are some other features that just one of the systems INtime or the offline scheduler maintains:

- INtime can identify user space tasks as real time processes whereas the offline scheduler cannot.
- INtime has a RealTime TCP-IP stack whereas the offline scheduler does not have.
- INTime can be used as a standalone RTOS whereas the offline scheduler cannot.
- INtime does not access Windows address space directly, whereas the offline scheduler accesses Linux address space directly.

The offline scheduler supports dynamic allocation of cores whereas INtime does not support.

C. IBM Logical partitions

IBM logical partitions [14,15] divide a server's resources into several logical units. The bus is shared; the memory and the disk storage may be partitioned and some other resources like communications can be also partitioned.

The partition model has a component similar to the offline scheduler; however as a concept the model is not relevant for the offline scheduler. In the offline scheduler system all the resources except the processor are shared. The offline scheduler can be depicted as a processor allocator;

there is no administrator. Each processor has an access to some resources in the system, as long as it does not call "schedule()".

The component that is similar to the offline scheduler is a primary partition of IBM that may be depicted as CPU 0 and if an offloaded processor fails, the system will continue to run if the operating system memory is not corrupted.

D. RTOP

Any Linux machine may hang sometime. Hanging might happen due to a Kernel crash or a System overload. What can be done when a server is remote? To detect a kernel crash "netconsole" can be used; however what can be done when the system is overloaded? The server cannot be accessed because it is so loaded that sshd (or any other daemon) does not get any cpu time. Utilities, vmstat, sar, top and so on, do provide details but under heavy load a user cannot access the machine remotely, or even locally. A user must be able to access the machine, and ask for the monitoring tool.

RTOP (Remote top) is monitoring software that provides top-like view of the system in any time. RTOP is based on the offline technology. This means that you must offline a processor (or enhance netconsole). RTOP is composed from RTOP utility ran in a remote Linux server and a driver that is invoked in the monitored server. RTOP driver pushes information out from the server. RTOP pulls information from the server.

III. THE OFFLINE SCHEDULER FOR REAL TIME OPERATING SYSTEMS

Vehicular systems are more often than not Real-Time systems [16]; nevertheless, there are also sometimes Non-Real-Time tasks [17]. In [18] the author explains the relationship between the new trend of SMP machines and the boost of the use of Linux versions as a Real-Time operating system. The offline scheduler suggests a new version of Linux scheduler for Multiprocessors real time operating system. The new scheduler splits the processes into two groups - the processes of the traditional operating system and processes of the offline scheduler. Actually, the offline scheduler is a hybrid system, because the new system is both real time and still a standard Linux server. The real time property is mainly achieved by the nmi and the CPU isolation characteristics; however the offline scheduler still interacts with the operating system.

Caches typically maintain these properties: spatial locality, temporal locality and sequentiality. Keeping these properties in usual scheduling environments is not easy. In the offline scheduler platform, they can be easily maintained. The offline scheduler gives the programmer tools to control the cache miss rate. Low cache miss rate has a harmful effect on performance [19,20]; hence the offline scheduler adapts some rules to handle the caching:

• Spatial locality

Accesses to same or near memory locations in the lifetime of a program are very likely. The offline scheduler abides this principle in its architecture. A user may assign a task to a processor without setting a different program into that cache. This way, the cache lines are not evacuated and the program data set and code are highly available.

• Temporal locality

Given a sequence of references to n memory locations, it is likely that following reference will be made into that sequence. The offline scheduler abides this rule same as spatial locality.

• Sequentiality

Given a reference to location n, it is likely that a reference to location n+1 will be made in the near future. Processors utilize this rule by pre-fetching data. Prefetching can be done in software or by the hardware. There is no true benefit in the offline scheduler but the fact that running nmi make sure no undesirable prefetches happen.

Traditionally, programs' size has been increasing [21] Nowadays, programs' execution code size usually exceeds the L1i size. Most parts of a real time program are characterized as non real time while a relatively small portion of it is real time. When a real time code shares cache storage with a management code, or IO code, it is flushed from the instruction cache (L1i) as well as its data (L1d and L2). The offline scheduler's architecture enforces a programmer to make the distinction between the real time codes to the rest. This distinction maintains the real time portion small.

If the working set is large enough, it is very likely that several memory locations will refer to the same cache line and cache flushes and misses may happen. The offline scheduler has no need to maintain memory pools like in [22]. It has access to almost all the memory (exceptions are vmalloc'ed/ioremap memory). A program having its own memory pool might cause congestion in cache lines if memory is not shuffled enough, but this will a fault of the Operating System; not of the offline scheduler.

Branch prediction is one of the processor's ways to speed up code execution. Problem with branch prediction is that a branch might get flushed too early. Brach flushing happens mainly due to interrupts, exceptions, segments descriptor loads [23]. The offline scheduler is running nmi so less pipeline flushes happen.

Multiple processors systems are usually designed for two reasons, faults tolerance and program speedup. Current Linux versions have little respect to fault tolerance processors Programs' speedup is achieved by a program design and not enforced by the operating system. The offline scheduler is a system where all processors share the same memory. The offline scheduler architecture forces the programmer to write a program with a better speedup because concurrent instances of the sub units of the program run in each processor. Fault tolerance is partially achieved by the fact the even if the operating system stops (panics) or hangs the offline scheduler can still run and vise versa, if an offloaded processor failure does not corrupt the operating system's active memory the system will not stop. This is obviously cannot be compared to a failure in some user space task, so in this sense running a user space task is better. Multiprocessors systems are subjected to the same restrictions as a single processor system, such as cache size, branch prediction, BUS speed, etc; which were discussed previously.

Partitioning is the process of dividing a task into sub-tasks, each of which can be assigned to a single processor. Partitioning a process is done in compile time. Partitioning objective is to reveal maximum amount of parallelism

possible without exceeding machine's hardware limitation [24,25].

A Partition analysis is preformed with some notion of a program overhead [18]. A program overhead (o) is the added time a task takes to be loaded into a processor prior to beginning execution. The larger the size of the minimum task, the smaller the partition overhead. The author of [21] describes program P work as follows: If a single processor P_1 program does operation O_1, the parallel version of P_1 does operations O_p where $O_p \geq O_1$. For each task T_i, there is an associated number of overhead operations o_i, so that T_i takes O_i operations without overhead then:

$O^T{}_p=\sum^p{}_i(O_i+o_i)\geq O_i$

Where O_p is the total work that was done by P_p including overhead, and $o_i+1 \geq o_i$. So, on one hand a better performance means maximizing parallelism and on the other hand finer code blocks increase the program overhead. Figure 1 depicts where the offline scheduler performs best.

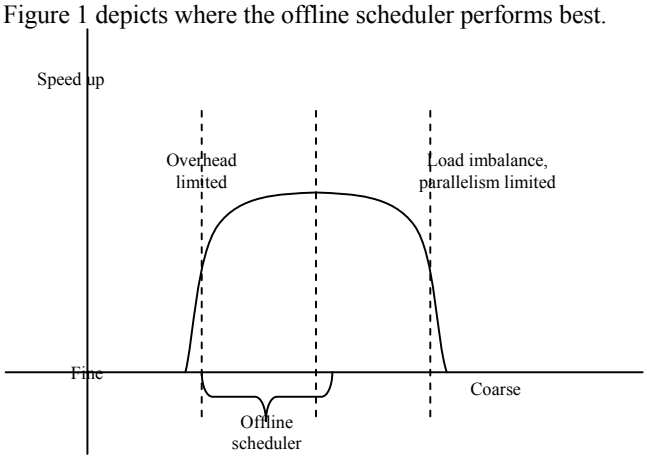


Figure 1: effects of grain size

At the offline scheduler, $o_i \rightarrow 0$, so there is no program overhead. This is because there is no stack manipulation, and all tasks are in the processor's cache. Load may be imbalanced only if several processors handle different resources and tasks. E. g., if one processor handles network interface A and a second processor handles network interface B, and the load on the two interfaces is not the same [26]. The offline scheduler cares less for load imbalance as long as the service is being done. If not, there is something wrong with the design. When load is well balanced, we can say that:

$O_i \approx O_1 \Rightarrow O^T{}_p=\sum^p{}_i(O_i+0)=O_{1p}$

Sometimes two tasks depends one on the other. In a single processor a typical scenario is:

1. T1 put message on T2 queue.
2. T1 schedule
3. Do something
4. T2 gets processor time
5. T2 replies on for T1
6. T2 schedules
7. Do something
8. T1 gets processor time
9. T1 gets T2's reply

The offline scheduler behaves very much the same. Yet, when using the offline scheduler, T1 may decide to schedule T2 directly, and only when T1 lets T2 will get a processor time; i. e. T1 chooses when T2 get processor time. Phases 7 and 3 do not exist in the offline scheduler. This small difference is extremely important when hard real time is required

Scheduling can be performed either dynamically or statically. Static scheduling is determined during compilation and dynamic is performed in real time. Ngai [27] observes that all major issues in runtime scheduling focus in insuring performance and reduce overhead losses. In multiple processors this is even a more important issue [28]. The following four run time scheduling major overheads include:

• Gathering information

Gather system and programs real time information. This includes, control structure, identification of critical paths and dependencies. Dynamic information includes work loads and resource availability. The offline scheduler has no need of gathering information. It is up to the designer to decide what information will be used. E. g. the offline scheduler timer uses "rdtsc" counters and Rdtsc has a very small overhead.

• Scheduling

In the usual course of events, scheduling is performed as shown in Table 1:

	Advantages	Disadvantages
Compile Time	Less overhead	Lack of fault tolerance Compiler lacks stall information
Run time	More efficient execution	Higher overhead

Table 1

When using the offline scheduler, scheduling is performed in run time and has a very little overhead. The offline scheduler has a single stack context [29]. The offline scheduler scheduling is a mere procedure invocation. There are two scheduling procedure in the offline scheduler:

- offsched _schedule

A routine is a registered in the offline scheduler pending queue. In the offline scheduler there is both fast scheduling, simplicity and hardly any overhead.

- The offsched reschedule

A routine for assignment of a task into offsched run queue (calendar). It is lockless and very simple.

• Dynamic execution control

This includes clustering and process creation at run time. In the offline scheduler, A process creation is the actual assignment to a designated processor (offsched schedule); there is no creation of a context as in the familiar UNIX. There are no pids, gids or similar features. There is no notion of clustering of processes.

• Dynamic data management

This includes a minimization of memory overhead delay when accessing data. Delay may improve by assigning tasks by a policy of a minimum memory delay. In the offline scheduler, memory access is matter good scheduling design. Due to the offline scheduler serialized nature, memory low latency scheduling policy can be easily achieved.

Consider the following scenario: Process A assigns a message to process B. Process A schedules process B after itself and exit. Process B runs right after and the memory is in the processor cache. When process B is completed, it will schedule process A and will exit. During this process, there is no cache miss as long as the data working set is smaller than the processor's cache.

IV. COMMUNICATION AND SYNCHRONIZATION

A typical programmer does not program when the processor cache or MESI (Illinois Protocol) in his mind. She does not know even on what hardware her program will be running on. So, her program is likely to have in-efficient memory accesses. If the program is intended to be running on multiprocessor machine, queues and synchronization primitives will be used. So cache misses, locks (atomic increment must seize the bus in a SMP system) and preemption are likely to load the system. The offline scheduler serialized nature reduces this contention.

One may ask, why not create a busy loop like:

```
While (can I do my thing)
        do my thing
```

This code will create a RCU starvation. RCU is a synchronization technique which enables multiple readers and multiple writers to access mostly accessed data structures from multiple processors. RCU starvation happens because each processor must walk through a quiescent state [30]. A quiescent state is when one of the bellow happens:

1. A processor performs a context switch.
2. A processor executes user mode.
3. A Processor executes the idle loop.

And neither of these will happen in a busy loop. The offline scheduler eliminates RCU starvation since the processor does not have to walk through a quiescent state as it is not part of the operating system.

Consider the following sequence of operations:

```
 INC RAX
ADD RCX, RBX
```

These instructions are independent. In regular pipelined processors these two instructions are not likely to run concurrently. Some commands might be associated with interrupts and exceptions. If the interrupt service routine runs in the initiating processor context, RCX and RBX content will be lost unless some additional logic is added to the processor [31]. Either case, a processor has to do additional work that reduces performance. The NMI property of the offline scheduler actually makes programs running in the offline scheduler context faster than a in an active processor. When designing multi-threaded software, one has to understand the price of a context switch with respect to the system requirements.

Moore Law presents us a simple statement, if in 2008 we have 4 cores in a single die, in 2010 we will be having 8 cores, in 2012 16 cores, in 2014 32 cores etc. Does the current Linux operating system design fits the Multi core era? Should the operating system handle so many cores? As this paper argues, the answer is: "not always". Having an operating system balancing tasks between 16 processors is not necessarily good. The Linux kernel migrate pages and move tasks between processors repeatedly to achieve balance in the system. The offline scheduler on other hand does not care much for imbalance; it aims for responsiveness, accuracy and simplicity. The offline scheduler can produce linear speed-up as long as it is not bounded by BUS speed.

V. THE OFFLET

In the offline scheduler environment, we do not use the terms threads, processes or softirqs. The offline scheduler refers to a processor as a device running an offlet. Offlet is the context of an offlined processor. When an offlined processor is removed from the operating system, It is simply set to the "halt" machine instruction while interrupts are disabled. Right before the halt instruction is executed, the offline scheduler is invoked. From now on, we are in an offlet context. Offlet context has some restriction that a user must be aware of:

1. A user may not cause any page fault, meaning he cannot access vmalloc'ed memory (only by "walking on the pages").
2. A user may not "STI".
3. A user may not invoke any code path that ends up in "schedule".

This is why the author refers to the offline scheduler context as an offlet; it is because it has more constraints than any other kernel context. offlets may be scheduled in time T, when T can be any time starting from now. Offlets may be setup on the stack, schedule and released; very much like any other kernel entity but much faster.

The offline scheduler is based on the notion that processors are not an expensive resource. In general, when task A is in kernel context and wishes to seize resource X all it does:

```
Step 0) ..
Step 1) Lock
Step 2) ..
Step 3) Unlock
Step 4) ..
```

If Step 4 is not serialized with respect to the previous steps then Offlet will suggest an asynchronous service, instead of spinning in step 1, the offline processor is asked to do steps 1 2 3 for the user. Offlet scheduling is very light so the scheduling cost is negligible in term of computation. Another added value of offlets is that a system may choose to serialize access to a device through an offlet. A good example is offline napi which is described below. This serialization actually relieves the kernel from contention on slow devices.

When any context enters the vmalloc area, it must generate a page fault in each processor this if the referenced address was not referenced before on the same processor. This is because vmalloc area is a dynamic mapped area and the processor's MMU has no reference to it. kmalloc address space is set on the kernel static page table that never change. For an offlet to access vmalloc area, or user pages, it must walk on the pages. Walking on the pages means dismantling the address to pages and calling kmap atomic on each page.

VI. THE OFFLINE TIMER

The offline scheduler timer is a plain busy-pause loop running at the frequency of the CPU (in our tests is has been 2 GHZ). Each time unit is divided into N slots; E.g. 1ms is divided into slots of 1000us. There is still a need to adjust the timer, because it uses the rdtsc mechanism which might be inaccurate. If a 100us resolution meets the requirements, HPET overhead will be quite small, about just 3% cpu usage in our tests (Lenovo T61 laptop). So, in such a case the offline scheduler timer will be redundant.

In cases when just few timer handlings are needed i.e. few timer handlings every several hundreds milliseconds, one can use TICKLESS [32] timer implementation.

However, since an operating system does other tasks, like processing packets, an interrupt might be nested and thus be

behind schedule. In addition, when the amount of timers is larger, the accuracy is less important.

VII. OFFLINE NAPI

Napi [33] stands for New API. Napi is a technique for interrupts mitigation in networking in the Linux kernel. NAPI technique is a simple polling over incoming packet arrival queue; this eliminates the constant interrupt processing. By default, Linux uses receive interrupt for packet processing, and activate NAPI when a certain threshold of number of incoming packets is reached. This threshold is known as the network "weight". Only one processor may call poll, this is because only a single processor can get the initial interrupt. There are some flaws of NAPI.

1. Latency. In some cases, NAPI may introduce additional software IRQ latency.

2. IRQ Masking. On some devices, changing the IRQ mask may be a slow operation, or requires additional locking. This overhead may negate any performance benefits observed with NAPI.

3. Rotting Packet. In some cases a packet may rot in the RX ring of the device. This happens due to a race between the time we decide we have no packet in the RX ring and the time we are forced to enable interrupts.

The offline scheduler NAPI is aimed to solve the above problems and provide infrastructure to other services for packet processing.

- Latency

Very much like NAPI, the offline scheduler NAPI will process incoming packets from RX ring. Unlike NAPI, spinning processors will have very little latency (if any) so disabling RX interrupts entirely is possible, this way we eliminate both problems. In addition, since we have no initial interrupt (an interrupt that triggers NAPI) we can have several processors spinning over the same device.

- IRQ masking

The duration of the operation of IRQ masking is not a problem because it is done only once. Once IRQ masking is complete, the offline scheduler NAPI is not bothered with it again.

- Rotting Packet

Because interrupts are never enabled and the operating system continuously spins over the network devices, this problem does no longer exist.

- SMP IRQ Affinity

Current IRQ affinity is in a device granularity. In offline NAPI, granularity may be based on service affinity. Meaning, instead of having all packets of a device routed to a group of processors, packets of type A may be assigned to processor X, type B to processor Y and so on. The rational here is that there is no sense in putting together ARP queries on the same processor as the user's application's processor.

VIII. EVALUATION

We have generated traffic through eth1 to eth1. Each machine has been connected to two different network segments. Traffic has been generated on the 10.9 segment.

Figure 2 depicts this test. The grey area is where the offline firewall process runs. It controls all interfaces. The test was conducted on a Supermicro PDMSI machine, with an Intel Pentium 3.4GHZ, Hyper threading enabled. Receiving interface was Intel 1Gbps 82546EB. e1000 and Linux kernel version was 2.6.30. The Loader is generating traffic Linux packet generator driver, also is known as pktgen. Pktgen UDP port is 9, which is the discard port.

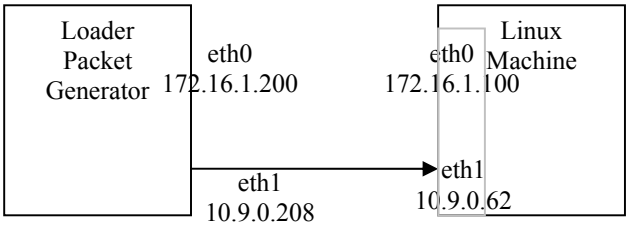


Figure 2: traffic configuration

```
ifconfig eth1  10.9.0.62/24
ping 10.9.0.208 -I eth1 -c 3
insmod offsched.ko
mknod /dev/offschedctl c 252 0
offschedctl -a 1
offschedctl -p 1  --setservice napi
offschedctl -i
insmod offlet_napi.ko
insmod offletnet.ko          ransmission driver
insmod filter.ko             firewall driver
offschedctl --start napi
insmod offlet_srv.ko saddr=172.16.1.62
daddr=172.16.33.203 dport=7777
sport=7777                   loads OFFSCHED
server on source address eth0 port 7777. The
destination address is not important; it is use
to get a routing interface pre-maturely.
```

Figure 3: Loading procedure

Figure 3 details the loading procedure. The test was first conducted with Linux Native NAPI. Then CPU1 was dropped and offsched was run.

```
Linux Native NAPI
Tasks:  66 total,   1 running,  65 sleeping,   0
stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%
wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,
65.3%id,  0.0%wa,  5.0%hi, 29.7%si,  0.0%st
Mem:   1025452k total,   153536k used,   871916k
free,     5084k buffers
Swap:        0k total,        0k used,        0k
free,    75568k cached

OFFLET SMT CPU1 is dropped and runs OFFSCHED.
Tasks:  58 total,   1 running,  57 sleeping,   0
stopped,   0 zombie
Cpu0  :  0.4%us,  0.4%sy,  0.0%ni,
92.9%id,  0.0%wa,  6.3%hi,  0.0%si,  0.0%st
Mem:   1025452k total,   194256k used,   831196k
free,    22528k buffers
Swap:        0k total,        0k used,        0k
free,    92808k cached
```

Figure 4:native NAPI vs. offlsched

Figure 4 shows the result of native NAPI vs. the offline scheduler. Using the offline scheduler, the hardware interrupts was 6% because receiving interrupts has not been disabled. When using Native NAPI, The consumption of CPU1 was 35%, whereas cpu0 is mostly idle. In the offline kernel CPU0 is 7% busy due to hardware interrupts.

The Intel network Interface was loaded with 1Gbps of incoming traffic as observed by bmon. In terms of pure processor consumption, 107/200 is the total processing consumption of offline NAPI while in native NAPI it is 35/200. So, one may wonder, are not we wasting a processor? The answer is actually yes, we do. But what we do earn is an operating system running undisturbed. If this machine has been loaded with more traffic, then the operating system may be too loaded to even be accessed. In a machine employing offline scheduler, this inaccessible state will not occur, because a denied packet processing is confined to the offloaded processor.

IX. CONCLUSION

Linux is a good platform for embedded systems [34,35]. Adding the offline scheduler will enhance the Linux abilities. The offline scheduler joins two different system types in a single machine, a Real Time system and a standard Linux machine. This is very helpful for systems like [2] where the authors use the embedded computer to monitor the train speed (a real time task) and to make sure the driver never opens the doors on the side without a passenger platform (a non-real-time task). Offloading processors to serve as dedicated engines might be very helpful in various real time scenarios. Using the offline scheduler, there is no need to buy expensive offloading network cards when the same feature can be achieved by commodity hardware. Who does need network processors for this feature, when every standard processor can do the same thing?

X. REFERENCES

[1] D. Geer, "Survey: Embedded Linux Ahead of the Pack," IEEE Distributed Systems Online, vol. 5, no. 10, pp. 3, Oct. 2004

[2] D. W. Carr, R. Ruelas, H. Salcedo-Becerra, and G. A. Ponce-Castaneda, "A Linux-based System to Monitor Train Speed and Doors for a Light-Rail System", Eight Real-Time Linux Workshop, Lanzhou, Gansu, China, October, 2006.

[3] Z Jianhua, Design of Electrical Monitoring and Control Termnals for Trains Based on Linux, Industrial Control Computer, 2005.

[4] V. Srovnal Jr, and J. Kotzian, Development of a Flight Control System for an Ultralight Airplane, International Multiconference on Computer Science and Information Technology, 745-750, 2008.

[5] Kepner, J. and Moore, M. and Travinin, N. and Kim, H. and Reuther, A. and Currie, T. and McCabe, A. and Mathew, B. and Rabinkin, D. and Rhoades, A. Deployment of SAR and GMTI signal processing on a Boeing 707 aircraft using pMatlab and a bladed Linux cluster, Technical Report, Massachusetts Institute of Technology, Lexington Lincoln Lab, 2004.

[6] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era" IEEE Computer, vol. 41 no. 7) pp. 33-38, July 2008.

[7] D. Eadline, "Multi-Core Melee", Linux Magazine, Issue #80, pp 40-41, July 2007.

[8] Dave Boutcher and Dave Engebretsen, "Linux Virtualization on IBM POWER5", Ottawa Linux Symposium, pp. 113-120, July 2004.

[9] S. Derr, CPUSETS, http://lxr.linux.no/linux+v2.6.26.3/Documentation/cpusets.txt, BULL SA 2004.

[10] P. Shinde, P. Sharma, S. Guntupalli, "Automated Process Classification Framework using SELinux Security Context," Third International Conference on Availability, Reliability and Security, 2008. ARES 08, pp.592-596, March 2008.

[11] "Solaris dedicated CPU, "http://docs.sun.com/app/docs/doc/8171592/gepsd?a=view

[12] W. Gentzsch, "Sun Grid Engine: Towards Creating a Compute Power Grid", pp.35, First IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), 2001.

[13] INtime, tenasys,http://www.tenasys.com/products/intime.php.

[14] "IBM Logical Partition Concept", http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.

[15] J. Jann, L. M. Browning, R.S. Burugula, "Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers", IBM Systems Journal, Vol. 42(1), pp. 29-37, 2003.

[16] Y. Wiseman, "Take a Picture of Your Tire!", Proc. IEEE Conference on Vehicular Electronics and Safety (IEEE ICVES-2010) Qingdao, ShanDong, China, pp. 151-156, 2010.

[17] I. Grinberg and Y. Wiseman, "Scalable Parallel Collision Detection Simulation", Proc. Signal and Image Processing (SIP-2007), Honolulu, Hawaii, pp. 380-385, 2007.

[18] P. E. Mckenney, " SMP and embedded real-time", Linux Journal, issue 153, p. 1, Jan 2007.

[19] H. Tomiyama, H. Yasuura, Code placement techniques for cache miss rate reduction, ACM Transactions on Design Automation of Electronic Systems, Volume 2 (4), pp. 410 - 429, Oct. 1997.

[20] U. Drepper. "What every programmer should know about memory" Red hat INC Copyrights 2007, http://people.redhat.com/drepper/cpumemory.pdf

[21] M. J Flynn. "Computer Architecture Pipelined and Parallel Processor Design", Jones and Bartlett Publishers Inc., 1995.

[22] M. Reuven and Y. Wiseman, Medium-Term Scheduler as a Solution for the Thrashing Effect, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 49(3), pp. 297-309, 2006.

[23] "IA-32 Intel Architecture Software Developer Manual Volume 3 System programming guide", http://pdos.csail.mit.edu/6.097/readings/intelv3.pdf. pp. 7-34, 2002.

[24] S. T. Klein and Y. Wiseman, "Parallel Lempel Ziv Coding", Journal of Discrete Applied Mathematics, Vol. 146(2), pp. 180-191, 2005.

[25] S. T. Klein and Y. Wiseman, "Parallel Huffman Decoding with Applications to JPEG Files", The Computer Journal, Oxford University Press, Swindon, UK, Vol. 46(5), pp. 487-497, 2003.

[26] Y. Wiseman, K. Schwan and P. Widener, "Efficient End to End Data Exchange Using Configurable Compression", Proc. The 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.

[27] T. F. Ngai, "run time resource management in concurrent systems", PhD thesis, Department of Electric engineering, Stanford university, January 1992.

[28] Y. Wiseman and D. G. Feitelson, Paired Gang Scheduling, IEEE Transactions on Parallel and Distributed Systems, Vol. 14(6), pp. 581-592, 2003.

[29] Y. Wiseman, J. Isaacson and E. Lubovsky, "Eliminating the Threat of Kernel Stack Overflows," in Proceedings of IEEE International Conference on Information Reuse and Integration (IRI 2008), pp. 116-121, Las Vegas, USA, July 2008.

[30] D. P.Bovet & M. Cesati. "Understanding the Linux Kernel". O'Reilly Media Inc.Copyrights 2006.

[31] D. Harry, "Computer organization for multiple and out-of-order execution of condition code testing and setting instructions", US Patent 5630157,1997.

[32] "kerneltrap", http://kerneltrap.org

[33] R. Bolla and R. Bruschi, "high-end Linux based Open Router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities", Proc. International Workshop on Internet Performance, Simulation, Monitoring and Measurements, pp. 203-214, 2005.

[34] J. Williams and N. Bergmann, Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip, Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, 2004.

[35] K.Yaghmour, J. Masters, P. Gerum and G. Ben-Yossef, Building embedded linux systems, O'Reilly Media, Inc., 2008.

PII

HYPLETS - MULTI EXCEPTION LEVEL KERNEL TOWARDS LINUX RTOS

by

Raz Ben Yehuda and Nezer Zaidenberg 2018

Proceedings of the 11th ACM International Systemsand Storage Conference.
2018

Hyplets - Multi Exception Level Kernel towards Linux RTOS

Raz Ben Yehuda
University of Jyväskylä
Jyväskylä , Finland
raziebe@gmail.com

Nezer Zaidenberg
University of Jyväskylä
Jyväskylä , Finland
nezer.j.zaidenberg@jyu.fi

ABSTRACT

This paper presents the concept of a Multi-Exception level operating system. We add a hypervisor awareness to the Linux kernel and execute code in hyp exception level. We do that through the use of Hyplets. Hyplets are an innovative way to code interrupt service routines under ARM. Hyplets provide high performance, security, running time predictability ,an RPC mechanism and a possible solution for the priority inversion problem. Hyplets uses special features of ARM8va hypervisor memory architecture.

1 INTRODUCTION

Available technologies today based on virtualization offer Microvisors that divide the computer into VMs, each VM encapsulating with its own hardware and has its own operating system. Many if not all of the above perceptions separates between higher exception levels to the lower exception levels.

This paper presents the hyplet ISR as a paravirtualization technique to reduce kernel to user latency to less than a microsecond on average.

We will also demonstrate a new RPC (Remote Procedure Call) functionality. Our RPC is a type of hypervisor trap where the user process sends a procedure id to the hypervisor to be executed in high privileges with no interruptions in another process address space. We use the term hypRPC for our RPC as a mixture between hypercall and RPC.

SYSTOR, 2018, Haifa, Israel

ACM ISBN 123-4567-24-567/08/06. . . $15.00
https://doi.org/10.475/123_4

Hyplets are based on the concept of a delicate address space separation within a running process. Instead of running multiple operating systems kernels in order to segment and divide the system resources, the hyplet divides the Linux process into two execution modes. One part of the process would execute in an isolated, non-interrupted privileged safe execution environment while other parts of the process would execute in a regular user mode. However, both execution modes run in the same Application processors.

We believe that Hyplets are suitable for hard real time systems. We will provide benchmarks and compare our solution to Linux RT PREEMPT and to a standard Linux. We chose RT PREEMPT as it is considered an open source non-commercial RTOS on Linux.

2 HYPLET ARCHITECTURE

In ARM, each of the four exception levels provides its own state of registers and can access the registers of the lower levels but not higher levels. This architecture dictates that the translation tables of the different exception levels are distinct. This means , that EL2 (hypervisor mode) may point to any memory translation table while the generic operating system, running in EL1 uses another translation table. This way, we can map an executing process (program) or part of it to the hypervisor, and we automatically gain access to the program address space, without any need to perform context switches or relocations.

To make sure that the hyplet code is always accessible and evacuation of the hyplet code and data from the current translation table is disabled, we chose to use $TTBR0\_EL2$ register to constantly accommodate the hyplet code. When a process address space is mapped to EL1 and EL2 excpetion levels it is reffered as a dual mapping of the process.

2.1 Hyplet - User Space Interrupt

In Linux, when an interrupt interrupts the processor, it triggers a path of code that serves the interrupt and in some cases ends up waking a pending process. The time

to wake up the process is the interrupt latency the hyplet reduces. To achieve this, as the interrupt reaches the processor, instead of executing the user program code in EL0 after the ISR, a special procedure of the program is executed in a hypervisor mode before the kernels ISR. This is possible due to the dual mapping. The hyplet does not require any special threads and should be implemented as a small procedure. Since hyplets are actually ISRs they can be triggered in high frequencies. This way we can have a high frequency user space timers in small embedded devices.

2.2 Hypervisor RPC

Interprocess communications (IPC) in real time systems is considered a latency challenge. One reason is because there is the possibility that the receiver is not running therefore the kernel needs to switch contexts, which is considered a penalty. IPC is also described as a possible priority inversion scenario problem. RPC (Remote procedure call) is a form of IPC in which parameters are transferred in the form of function arguments and response is returned in the form of function return value. The RPC mechanism handles the parsing and handling of parameters and return values. We will show that it is possible to define a procedure in the address space of a receiving process that is invoked as a callback through the hypervisor whenever a sending process triggers an RPC. In this paper hyp RPCs are a form of IPC, i.e they are local.

3 EVALUATION

All tests were performed on Raspberry PI 3.

3.1 Interrupt Latency

We tested Hardware to hyplet latency. The Interupt latency was 2.5μs. Evidently,hyplets have a low latency and also are suitable for hardware that generates interrupts at different rates.

3.2 Timer

We compared the responsiveness of hyplets to Linux RT PREEMPT.

In the hyplet case, 99.96 % of the samples were bellow 1μs latency, and 100% were bellow 5μs. In RT PREEMPT case, the upper boundary is 47μs and the average is over 14μs. It is evident that ISR-hyplets provide hard real time in a regular kernel.

3.3 Fast RPC

We evaluated the round trip of calling a null function (it just returns the time). A common IPC usually involves two context switches in a full round trip. The below benchmark is between two processes, a receiver and a sender. The receiver maps a hyplet to a single core, and the sender calls it. There are four types of tests:

- **Ref** Duration of the function when called in the process.
- **Hyplet** Duration of the function when called by a hypRPC and the sender and receiver share the same core.
- **IPI-hyplet** Duration of the function when called by a hypRPC when the sender and receiver do not share the same core
- **Standard Linux** The sender and receiver exchange is undertaken by Posix semaphores. The receiver waits on a semaphore; the sender awakes it and then waits on a second semaphore; the receiver executes the null function, and releases the sender.
- **Linux RT_PREEMPT** Like Standard Linux but over RT PREEMPT.

	Min	Avg	Max
Ref	104ns	156ns	520ns
Hyplet	520ns	520ns	4.2μs
IPI-hyplet	3.4μs	6μs	21μs
Normal Linux	2.3μs	102μs	208μs
RT PREEMPT	12μs	14μs	340μs

It is evident that hyplets are the fastest.

4 OTHER FEATURES

4.1 Security

We provide a safe execution environment for the operating system kernel so that even if there is a fault in the hyplet or if malicious data (that somehow crashes the hyplet) arises in the computer we can choose not to stop (or panic in Linux terms) the operating system. This is possible because the fault entry in EL2 handles the error as if it is a user space error. We can access hardware and protect its data. This can be done by reading data into a secured memory that is not accessible from EL1.

4.2 Temporality

Interrupts service routines rarely change ,i.e. it is not easy to modify a behavior of an interrupt routine in real time (while the device is running) .In the hyplet case, instead of modifying the kernel drivers, we can kill the user space hyplet program and run a new hyplet. We also have an abort mechanism that protects the hyp-ISR from crashing the operating system when there is a failure. The hyplet will fault like any other user space task.

PIII

ARM SECURITY ALTERNATIVES.

by

Raz Ben Yehuda, Roee Leon and Nezer Zaidenberg. 2019

Proceedings of the European conference on information arfare and security.Academic Conferences International. 2019

ARM Security Alternatives

Raz Ben Yehuda[1], Roee Leon[1] and Nezer Zaidenberg[2]
[1]University of Jyväskylä, Finland
[2]College of Management Academic Studies, Rishon LeZion, Israel
rabenyeh@student.jyu.fi
roee@trulyprotect.com
scipio@scipio.org

Abstract: Many real-world scenarios such as protecting DRM, online payments and usage in NFC payments in embedded devices require a trustworthy "trusted execution environment" (TEE) platform. The TEE should run on the ARM architecture. That is popular in embedded devices. Furthermore, past experience has proved that such TEE platform should be available in source code form. Without the source code 3rd parties and user cannot be conducted code review audit. Lack of review put doubt on the system as a trustworthy environment. The popular Android OS supports various TEE implementations. Each TEE OS implementation has its own unique way of deploying trusted applications(trustlets) and its own distinct features. Choosing a proper TEE operating system can be a problem for trust applications developers. When choosing TEE applications developers has many conflicting goals. The developers attempt to ensure that their apps work on as many different Android devices as possible. Furthermore, developers relay on the TEE for certain features and must ensure the suggested TEE provides all the features that they need. We survey multiple ARM TrustZone TEE operating systems that are commonly available and in use today. We wish to provide all the information for IoT vendors and SoC manufacturer to select a suitable TEE.

Keywords: virtualization, ARM architecture, TrustZone, trusted computing

1. Introduction

The proposed solutions for creating TEE on the ARM architecture are all using TrustZone™ feature. TrustZone™ is a unique privilege level on ARM (ARM 2009) whose purpose is to create a Trusted Execution Environment (TEE) (Zaidenberg 2018). Trustzone™ can be found on virtually all modern mobile phones. Additionally, TrustZone™ can be found in other ARM based systems on chip, such as AMD with their "Hiero falcon", AppleMicros X-Gene3, Cavium Thunder X and other systems.

Trust Execution Environment is required on many scenarios. TEE use cases include providing Digital right management (DRM)support. DRM requires a root of trust that can store decryption keys on the end point so that it will not be available to outsiders (Zaidenberg et al 2015b). Using virtualization to create root of Trust was attempted by SONY on PS4 and later shown on PC and MIPS by (Averbuch et al 2013). A complete system for reverse engineering protection by distribute keys for encrypted code execution after attestation (Resh et al 2017) introduced the concept of protecting code and registers by the hypervisor. (Kiperberg et al 2019) proposed creating buffers of protected code in the CPU case using memory addresses that only the hypervisor can address. Virtualization can also provide end point security (Resh et al 2017). Hypervisor can also be used for development aid by catching hypercalls (Khen et al 2011) connecting virtual debug hardware (Khen et al 2013) forensics (Zaidenberg et al 2015) and Forensic memory dump (Kiperberg et al 2019b). Last hypervisor can be used for end point attestation as shown by (Kiperberg et al 2013, Kiperberg et al 2015) etc.

Using virtualization as a tool for cyber security is also a common practice. Virtualization was initially designed for dynamic provisioning of computing resources. However, virtualization offers higher privileges and execution permissions that can used to detect threats as well as serve as TEE. ARM didn't offer virtualization or TrustZone™ support until the ARM7a hardware. ARM virtualization and TrustZone™ technologies were proposed as two optional additions. These technologies are available in some 32bit ARM7a models. ARM virtualization and TrustZone™ are now part of the 64bit ARM8a architecture and offered in all ARM8a devices.

ARM vendors offer their own TEE implementations. Some TEE implementations such as Trustonic's TSP and Qualcomm's QSEE are closed source while other are open source or offer providing the TEE source code for a fee. We survey Trusted computing alternatives for implementations. We mainly consider alternatives with available source code. All the surveryed solutions offers a complete solution for the TrustZone™ environment. We also survey some ARM virtualization (not TrustZone™) based alternatives.

2. Background

2.1 Trusted Execution Environment

The ARM architecture design allows both a Trusted Execution Environment (TEE) and a Rich Execution Environment (REE, i.e. normal ARM OS e.g. Android or iOS) to run simultaneously. The Trusted Execution Environment is a secure "area" inside a main processor not effected by the REE operating system. The Trusted Execution Environment runs its own operating system and uses its own set of register and its own memory management unit (MMU). The TrustZone™ operating system is a separate operating system that is running in parallel to the main operating system in an isolated environment. The Trusted Execution Environment guarantees that the code and the data loaded in the TEE are protected with respect to confidentiality and integrity from all application that run on the REE OS.

The Rich Execution Environment is a separate privilege level inside the main processor. The Rich Execution Environment runs its own separate operating system (compared to TEE). The Rich Execution Environment is the standard operating system that the device is running, usually the REE is Google's Android or Apple's iOS. The Rich Execution Environment offers significantly more features and applications than the TEE. This is by design and application are supposed to run on the REE and not on the TEE. As a result of offering more features, the attack surface against the REE is much larger and therefore, the REE is more vulnerable to attacks. The Rich Execution Environment receive services such as decryptions and storing decryption keys from the Trusted Execution Environment. According to ARM design the TEE acts as a monitor service for the REE.

The TEE has higher permissions than the REE as well as access to the REE memory, MMU, registers and data structures. The REE should not have access to the TEE memory and data structures. In ARM terminology, the two execution environments are called worlds, the secure world (TEE) and the non-secure world(REE). Context can be switch between secure and insecure worlds through the supervision the "Secure Monitor" running in monitor mode(TrustZone). This switch from secure to insecure world is performed through a special architecture specific assembler instruction called "secure monitor call" or smc. In order to communicate between the secure and non-secure world the user creates a shared memory segment. TrustZone™ splits the SoC device to the secure and non-secure worlds. TrustZone™ controls all the device hardware interrupts. TrustZone™ can route any interrupt to the secure world or to the non-secure world. Like in the memory case, I/O and interrupts routing may change dynamically. TrustZone™ uses its own MMU. Operating systems and Processes that execute in TrustZone™ do not share the same address space with their non-secure world counterparts. Thus, there is no need to have distinct TrustZone™ for each processor. A single TrustZone™ OS can run across multiple ARM processors/cores and manage all the device trusted computing needs. The ARM architecture cryptographic keys are accessible only in TrustZone™, The manufacturer can provide each CPU or platform with device specific keys using e-fuses. These keys are device specific, thus enabling protection in the end unit granularity level.

(For example distributing video that only specific device can decode etc.)

Booting a Trusted Execution Environment must form a chain of trust in which a trust nexus verifies the next component on the boot chain. Each component verifies the next component until the system.

2.2 ARM permission model

The ARMv8 architecture has a unique approach to privilege levels. The ARM platform normally has 4 exception (permission) levels.

ARM also has secure world (TrustZone™) and normal world (non TrustZone™)

ARM Exception levels are described in Table 1 Each of the exception levels provide its own state of special purpose registers and can access these registers of the lower levels but not higher levels. The general-purpose registers are shared.

Thus, moving to a different exception level on the ARM architecture, does not require the expensive context switch that is associated with the x86 architecture.

Table 1: Arm exception levels

Exception level	Meaning	Notes
Exception Level 0 (EL0)	Refers to user space code.	Exists in both secure and normal world This is analogous to "ring 3" in x86 platform.
Exception Level 1 (EL1)	Refers to operating system code.	Exists in both secure and normal world This is analogous to "ring 0" in x86 platform.
Exception Level 2 (EL2)	Refers to HYP mode. ARM hypervisor privilege level	Exists in both secure and normal world This is analogous to "ring -1" or "real mode" on the x86 platform.
Exception Level 3 (EL3)	TrustZone™	Refers to TrustZone™ as a special security mode that can monitor the ARM processor and may run a security real time OS. There is no direct analogous modes but related concepts in x86 are Intel's ME or SMM. Naturally it exists only on secure world

ARM7 architecture is similar to ARM8. ARM7 offers virtualization as an extension that is only available to some late ARM7 models. ARM7 also offer TrustZone™ as separate extension. Furthermore, ARM7 is 32bit architecture while ARM8v is 64bit (and 32bit) architecture.

3. Virtualization vs. TrustZone™ mode

The first question we must address is how the operating system should be verified. The REE operating system can be verified using HYP mode or TrustZone™. ARM has designed the TrustZone™ mode specifically for attesting and monitoring the Rich operating system. The benefit of using TrustZone™ is that it reserves the HYP mode for real hypervisor without the need to use features such as nested virtualization. Furthermore, only the vendor can install software on the TrustZone™ mode. In some cases, even the vendor (i.e. the manufacturer of the device or phone, not the CPU vendor) has limited access and cannot install software in TrustZone™ mode. However, no such limitation exists on HYP mode. Everybody can install software in HYP mode with no special limitations. This usually makes hypervisor code easier to install. From the device vendor standpoint therefor it is assumed that TrustZone™ is available. (or possibly available) From software vendor standpoint TrustZone is not available but virtualization may be.

The two main drawbacks of using virtualization are that virtualization mode is no longer available for other software that may want to run there. Also, TrustZone™ is monitored on boot by the BSP (Board Support Package) it cannot be modified or replaced as easy as the hypervisor boot loader or driver.

Resh et al (2017) and Seshadri et al (2007) both provide examples of using hypervisor for end point security.

We examine several hypervisor implementations for completion However it is assumed that a TrustZone™ solution is preferable whenever TrustZone™ is available.

3.1 Virtualization classification

Virtualization is the process of running multiple Operating system on a single hardware or running microkernel to manage single operating system. Hypervisor is the software that provides virtualization. Hypervisors are classified to two modes:

- 1. Full virtualization - The guests operating system is not modified in any way.
- 2. Para-virtualization – The guest operating system is aware it run as guest. The guest's operating system code is modified. The guest operating system does not attempt to communicate directly with the hardware. Instead the guest operating systems uses hypercalls to communicate with the host hypervisor. When the guest's operating system needs the host for example for I/O access and sometimes in critical sections. The hypercalls trap to the hypervisor to perform a service on behalf of the guest. Using para-virtualization and hypercalls usually yields better performance.

In the taxonomy of virtualization environment, virtualization environments are categorized by their design.

- 1. Complete monolithic - A single software responsible to provide access to the hardware to the guests. For example, VMware ESXi server.

- 2. Partially monolithic - The technology is an extension to the general-purpose operating system, such as KVM in Linux and VMWare Desktop or Microsoft Hyper-V
- 3. MicroKernel These are light weight micro kernels that a minimal set of services to the guests, mainly CPU virtualization and hardware access. Xen and seL4 are examples for such micro hypervisors.

Last virtualization pioneers Popek and Goldberg (Popek et al 1974) classified hypervisors to type I and type II

Type I hypervisors (or boot hypervisors) – are hypervisors that start at boot and start various guest operating systems. Examples include VMWare ESXI, Xen and IBM S/390 VM.

Type II Hypervisors (or hosted hypervisors) – are hypervisors that starts under a host operating system that already booted and took control of the machine. Examples include VMWare desktop.

For security and trusted computing purposes only Type I hypervisors are of interest. Type II hypervisors can be disabled by the host OS and thus serve no security purpose.

4. Alternatives review

4.1 GlobalPlatfrom

GlobalPlatform is a non-profit organization that consist of an alliance of many mobile device manufacturers. GlobalPlatform defines and publishes the standards for mobile devices including a standard for secure digital services for mobile devices. GlobalPlatform (2011) is the current industry standard for TEE platform under ARM.

4.2 General Dynamics OKL4

OKl4 is a microkernel that was originally developed, maintained and distributed by Open Kernel Labs.

The OKL4 operating system was based on the L4 operating system by Liedtke (Liedtke 1996).

The L4 microkernels family in its earlier form was called L3. L3 was a microkernel that was developed Liedke in the 1980's on an i386 system and was deployed in few thousands' installations, mainly education institutes. L3 suffered from a high overhead of Inter-process communication (IPC) communication which was over 100us. Liedtke, trying to reduce the IPC overhead problem, had re-implemented L3 completely and reduced significantly the IPC overhead to 5us, on i486. This new design was referred as L4.

L4 had evolved over the years and become a family of L4 microkernels, to name a few, L4-embedded, Codezero, NICTA, sel4 etcetera. NICTA was maintained by OpenLabs which renamed it to OKL4 microkernel and stopped the open source development.

OKL4 (Heiser et al 2010) is deeply discussed in peer reviews press. The OKL4 micro-visor supports both paravirtualization and pure virtualization. It is designed for the IoT industry, and supports, ARMv5, ARMv6, ARMv7ve and ARM8va. OKL4 is focused on embedded devices. OKL4 was originally open source software.

On 2012 general dynamics acquired the Open Kernel Labs. After being acquired General Dynamics changed OKL4 source code policy from open to closed source project. The latest available open source OKL4 is from May 2013 and is still available to download from archive.org (and other sources). OKL4 has a sister open source project (supported by GeneralDynamics) called seL4 which is described later.

Installing OKL4 and running it is a challenging task that requires expertise. Open source OKL4 code must also be adapted to modern hardware. Today OKL4 is still under development and support of General Dynamics.

One can also obtain the current OKL4 source code under suitable license and NDAs.

We refer to the latest available open source OKL4 from 2013 and not to current releases (for which source code is not available) thus we are not up to date with current releases (compared to other Trusted Execution Environment alternatives).

4.3 Google Trusty TEE

Trusty is a secure Operating System (OS) that was developed by Google.

Trusty provides a Trusted Execution Environment (TEE) for The Android (only) Operating system.

The Trusty OS doesn't require security specific hardware. Instead, Trusty TEE runs on the same processor as the normal Android OS, However, despite running on the same CPU, Trusty is isolated from the rest of the system. This is done using ARM TrustZone™ features that enable separate MMU for trusty (in TrustZone™) and the normal world OS. TrustZone™ allows Trusty to create an isolated secure execution environment and provide certain services to the non-secure (i.e. Android) OS.

Trusty consists of:

- A small operating system kernel. The trusty Kernel is derived from Little Kernel. Little Kernel is a small operating system that is also used as Android boot loader.
- A Linux kernel driver that acts as mediator between the TEE (Trusty) and REE (Android) environments
- An Android user space library that provide to communication between the REE (Android) and TEE (Trusty) applications using the kernel driver

Trusty is compatible with ARM and Intel processors. On ARM systems, Trusty uses ARM's Trustzone™ to virtualize the main processor and create a secure trusted execution environment. Similar support is also available on Intel x86 platforms using Intel's Virtualization Technology.

4.4 Linaro OP-TEE

Linaro security working group and STMicroelectronics have teamed to create OP-TEE.

OP-TEE follows GlobalPlatfrom specification (2011) and implements version 1.1 of GlobalPlatform TEE client API and TEE internal API. OP-TEE is an open source project and is widely available under BSD 2-clause license and (Kernel parts) GPLv2 license. According to (Bech 2014) and our testing OP-TEE has a small foot print and minimal effect on the running system. OP-TEE has a large community support. Like Trusty above OP-TEE consists of 3 main components.

- A light weight secure operating system. The OP-TEE operating system consists of several modules such as memory management, interrupt handling, etc. In addition, OP-TEE implements a hardware abstraction layer as it supports various processors and hardware. The OPTEE operating system also provides a capability to run user-space applications (typically referred to as Trustlets) in the secure world. These applications are provided with the GlobalPlatform TEE Internal API which allows them to ask for internal, secure-only, OS services
- A non-secure user-space client that is composed of two components: (1) a user-space/kernel-space mediator and (2) libraries that implement the GlobalPlatform TEE Client API.
- A kernel driver that simply performs the transitions between the secure and non-secure worlds

5. Other alternatives

There are other TEE options that the vendor can use for both EL2 and EL3. These alternatives are not as common or as strong as the alternatives mentioned above and fail in atleast one pre-requisite.

5.1 Jailhouse

Jailhouse was announced by Siemens in November 2013. Baryshnikov (Baryshnikov 2016) has analyzed the Jailhouse system. Jailhouse is a type 2 partitioning microvisor for Linux hosts.

A partitioning microvisor is a microvisor that controls the OS access to resources and isolates the resources from the general-purpose operating system or other guests. Partitioning in microvisor context means the microvisor performs strict allocation of the system resources. The hosting Linux is referred as the Root cell, and the guests are called inmates. Jailhouse itself is not an operating system, it is a resource access controller. Jailhouse is

controlled from the Linux host, and reveals information stored to the Linux host (the root cell), but not the guests (the inmates).

Jailhouse is a bare metal hypervisor, and in most cases, it is pure virtualization hypervisor, and as such can run many types of operating systems, such as FreeRTOS (Barry 2008), Erika3 (Evidence 2019)``, Linux and Zephyr. Jailhouse supports ARMv8, ARMv7a, and x86_64 architectures. Jailhouse requires the machine to have at least two processors. One processor is used to run the hosting Linux, and the other processors may be assigned to Jailhouse. Jailhouse requires the Linux kernel to provide a contiguous memory at boot time. It requires a memory footprint of few tens of megabytes, usually 50 megabytes.

The Jailhouse configuration is performed through a tool provided by Jailhouse. This tool scan sysfs and procfs, and generates a device tree that describes the hardware as seen by the Linux host. This Jailhouse device tree is referred as the cell configuration file. The user can edit the cell configuration file to create a correct guest configuration. Jailhouse targets the automation, robotics and IoT industries.

5.2 QSEE

QSEE is Qualcomm Secure Execution Enviornment. In the past it was based on OKL4 until GeneralDynamics and Qualcomm failed to reach agreement regarding licensing. Since 2015-6 Qualcomm has developed QSEE from scratch with no (direct) connection to GeneralDynamics.

QSEE is closed source (and Qualcomm does not provide source code licenses) Therefore QSEE fails our precondition of source availability and is not part of this survey. Prior releases of QSEE suffered from several well documented security problems.

(Beniamini 2016)

5.3 seL4

sel4 like OKL4 is also a based on the L4 microkernel. seL4 (Klein et al 2009) is a microvisor that was implemented by Open Kernels labs (and later GeneralDynamics). seL4 is not as popular as OKL4. One of the strong features of seL4 is the fact that it has been formally verified to be correct. The method of verification is definition of seL4 exact functional specification and proving its operations using rigorous logical means. Later the seL4 codebase was reimplemented in C to make it efficient. Despite the rigorous testing seL4 is not necessarily bug free. The implementation has some assumptions of correctness about the compiler, architecture and C reimplementation.

seL4 compared to OKL4, is an open source kernel. seL4 is the sole kernel that is mathematically proven secured and safe. L4-embedded, or NICTA embedded, was adopted by Qualcomm as a real time operating system for their wireless modem processors firmware.

The basic rules of the L4 kernel design are minimalism. Leidtke (1996) formulated the rule of minimization as follows:

> *"A concept is tolerated inside the u-microkernel only of moving it outside the kernel, i.e. permitting competing implementation would prevent the implementation of system required functionality".*

This principle, known also as the no-policy in the kernel is the core of the L4 microkernels design. Though operating systems tends to grow in size over the years, L4 footprint is considerably low. seL4 footprint is 9600 lines of code. As a side effect of the minimization and performance, L4 microkernels, do not strive to hardware abstraction. Half of the seL4 microkernel is agnostic to the underlying hardware.

L4 also demonstrates a new resource management scheme where all memory allocations are user space driven. Another interesting feature of the L4 microkernels, is the fact that interrupts are disabled while executing in kernel mode. This approach simplifies the implementation, increases the performance and eases the kernel verification. Direct process switch, which in general means that sel4 tries to avoid from using the scheduler, is another interesting facet of L4. When a thread reaches a pre-emption point, the kernel switches to the first runnable thread, which in turn, executes on the time slice of the pre-empted thread.

seL4 runs on ARM, supports SMP and Uniprocessor. Like OKL4 seL4 also provides real-time support.

seL4 was recently ported to ARM8 architecture (Heiser 2019)

5.4 TrustTonic

TrustTonic is known for its TrustZone technology in the mobile world, mainly Android. TrustTonic operating system, Kinibi, is a closed source operating system. Kinibi is wide spread in the Android cellphone world. Kinibi provides data encryption and device authentication. It also gives safe access to peripherals, such as the touch screen, NFC and finger print reader, through its TEE. Since peripheral I/O is provided using the TEE no malware in the REE will affect the I/O. In addition, Kinibi can isolate sensitive code execution and secure data.

Kinibi is verified by the chain of trust, i.e; it is verified by the bootloader each time the device boots.

Furthermore, TEE application has access to the network. This way, a trusted application can access remote services securely.
TrustTonic can also be found in the automotive industry. In this area, TrustTonic approaches data leakage, application overlapping and application re-packaging attacks. Application overlapping attack is an interception technique for stealing sensitive I/O, such as when a user enters its password. A repacking of an application is a method of modifying a program to steal sensitive data. For example, adding a log entry that prints sensitive information.

Trustonic offers an SDK, compliant with GlobalPlatform API standards, to help build Trusted Applications.

Since no open source version of Kinibi exists (not even older version) we left it out of this survey.

5.5 Xen

Xen was announced in 2003. Xen was developed initially at the university of Cambridge, by Ian Pratt. Xen is a micro kernel hypervisor. Xen provides CPU virtualization using virtual interrupts,MMU and under-guest communication. In Xen, a virtual machine is referred as Domain. Domain0, also known as Dom0, is the first domain. Dom0 must run before any other virtual machine. Dom0 is usually Linux or BSD, and DomU is a virtual machine on top of the other domains. Domain0 requires access to the entire machine's hardware. Domain0 responsibility is management through the Linux kernel. Xen's event channel provides communication between Dom0 and DomU. Whenever DomU issues a virtualized event it uses this event channel. The event channel is used for para-virtualized guests. For a full virtualized guest Xen uses QEMU. Xen's tool stack is the management tool used to control guests. The fact that Xen uses Linux as Dom0 provides Xen with abundant of hardware support and Linux software. Xen boots from the bootloader and is then loads the a para-virtualized host.

Xen's I/O virtualization comes with a performance cost. Virtualized I/O accesses and virtualized interrupts from DomU Xen guests are delegated to Dom0. In addition, if a host interrupt occurs while DomU runs, then this interrupt would be served only when Dom0 is gets the processor. Thus, interrupts and events have a performance overhead.

Xen is available in ARM and x86, runs on SMP and UP. Xen is licensed GPL.

5.6 Xvisor

Xvisor was announced in Apr 2012. Xvisor (Patel et al 2015), is a monolithic, type 1 hypervisor that is independent of Linux. Xvisor is a monolithic hypervisor that controls the hardware peripherals. Xvisor provides a minimal operating system, thus Xvisor is not a microkernel. Xvisor can emulate devices and provides a path-through access to real devices. As an operating system, Xvisor has a memory management, scheduler, load balancer and threads. Xvisor does not support processes and is not POSIX compliant. Xvisor supports SMP, so that a guest can use two or more processors. There are no restrictions on the number of processors, and XVisor can also execute on a single processor.

Xvisor provides an IPC between two guests through the use of aliased guest region, which is

a GPA (guest physical address) shared between two guests. In the Xvisor taxonomy, a processor can be Normal VCPU or an Orphan VCPU.

Normal VCPUs serve guests OSes, and Orphan VCPUs belongs to the hypervisor. Xvisior support ARM 32bit and 64bit and x86. Its footprint is less than 10MB, however, since it is a type 1 hypervisor, it is required to change the boot loader. Xvisor is widely targets the infotainment market in the automobile world, and automation in general. It is licensed GPL.

6. Discussion

We surveyed most well known TrustZone and HYP alternatives. If the user requires open source and community review as security requirements then both trust-tee and OP-TEE provide good alternatives.

OP-Tee provides the benefit of following GlobalPlatform standard and is more light weight and provide less features than google Trust-Tee. OP-Tee has been an open source OS but the source is only available to paying customers. However, OP-Tee (and seL4) offer real time support that are required for some systems. We also reviewed seL4 which is also open source. seL4 is an option but is not as widely used as OP-Tee and Trust Tee (or OKL4).

Furthermore, TrustZone™ and other extensions can be used for other means such as real-time processing (Ben Yehuda et al 2018) and control flow analysis (Abera et al 2016). Using the TrustZone™ architecture for other purposes is an interesting future research area

7. Conclusion

We surveyed the popular TEE alternatives available today. Each alternative has its own benefits and drawbacks. SoC vendors and Platform manufacturers can choose the desired implementation based on their requirements and preferences. Out of the popular alternatives we believe that the free alternatives Google TrusTEE and OP-TEE offer sufficient features and fair replacement for OKL4. We believe OKL4 has none security benefits in strict real time environments that are beyond the scope of this review.

References

Abera, T., Asokan, N., Davi, L., Ekberg, J. E., Nyman, T., Paverd, A., Sadeghi A.R & Tsudik, G. (Abera et al 2016). C-FLAT: control-flow attestation for embedded systems software. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 743-754). ACM.

Averbuch, A., Kiperberg, M., & Zaidenberg, N. J. (Averbuch et al 2013). Truly-protect: An efficient vm-based software protection. *IEEE Systems Journal*, *7*(3), 455-466.

ARM, Architecure (ARM 2009). Security technology building a secure system using TrustZone™ technology (white paper). ARM Limited.

Barry, R. (Barry 2008). FreeRTOS. *Internet, Oct*

Baryshnikov, M. (Baryshnikov 2016). Jailhouse hypervisor (Bachelor's thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum.).

Bech, J. (Bech 2014). OP-TEE, open-source security for the mass-market. Core Dump. https://www.linaro.org/blog/op-tee-open-source-security-mass-market/

Ben Yehuda R. and Zaidenberg N. J (Ben Yehuda et al 2018) Hyplets - Multi Exception Level Kernel towards Linux RTOS In the Proceedings of the 11th ACM International Systems and Storage Conference Systor 2018 pp 116-117

Beniamini G. (Beniamini 2016) "Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption". https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html

Evidence Srl (Evidence 2019) "ERIKA Enterprise 3 source code" https://github.com/evidence/erika3

Heiser, G., & Leslie, B. (Heiser et al 2010). The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In Proceedings of the first ACM asia-pacific workshop on Workshop on systems (pp. 19-24). ACM.

Heiser, G (Heiser 2019) "Whats new in the world of seL4" FOSDEM 2019 https://www.youtube.com/watch?v=6s5FDX5PkZI

Khen, E., Zaidenberg, N. J., & Averbuch, A. (Khen et al 2011). Using virtualization for online kernel profiling, code coverage and instrumentation. In *2011 International Symposium on Performance Evaluation of Computer & Telecommunication Systems* (pp. 104-110). IEEE.

Khen, E., Zaidenberg, N. J., Averbuch, A., & Fraimovitch, E. (Khen et al 2013). Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation. In *2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (pp. 78-85). IEEE.

Kiperberg, M., & Zaidenberg, N. (Kiperberg et al 2013) Efficient Remote Authentication. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*(p. 144). Academic Conferences Limited.

Kiperberg, M., Resh, A., & Zaidenberg, N. J. (Kiperberg et al 2015). Remote Attestation of Software and Execution-Environment in Modern Machines. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (pp. 335-341). IEEE.

Kiperberg M, Algawi A, Leon R, Resh A. & Zaidenberg N. J. Hypervisor-assisted Atomic Memory Acquisition in Modern Systems (Kiperberg et al 2019b) in Proceedings of 5th international conference on information system security and privacy ICISSP 2019

Kiperberg, M., Leon, R., Resh, A., Algawi, A., & Zaidenberg, N. J. (Kiperberg et al 2019). Hypervisor-based Protection of Code. *IEEE Transactions on Information Forensics and Security*.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe D., Engelhardt K., Kolanski R. ,Norrish M., Sewell, T., Tuch H. & Winwood S. (Klein et al 2009) . seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (pp. 207-220). ACM.

Liedtke, J. (Liedtke 1996). Toward real microkernels. Communications of the ACM, 39(9), 70-77.

Patel, A., Daftedar, M., Shalan, M., & El-Kharashi, M. W. (Patel 2015). Embedded hypervisor xvisor: A comparative analysis. In Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on (pp. 682-691). IEEE.

Popek, G. J., & Goldberg, R. P. (Popek et al 1974). Formal requirements for virtualizable third generation architectures. Communications of the ACM, 17(7), 412-421

Resh, A., Kiperberg, M., Leon, R., & Zaidenberg, N. (Resh et al 2017b). System for Executing Encrypted Native Programs. *International Journal of Digital Content Technology and its Applications*, *11*

Resh, A., Kiperberg, M., Leon, R., & Zaidenberg, N. J. (Resh et al 2017). Preventing Execution of Unauthorized Native-Code Software. International Journal of Digital Content Technology and its Applications, 11.

Zaidenberg, N. J. (Zaidenberg 2018). Hardware Rooted Security in Industry 4.0 Systems. *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures*, *51*, (pp 135-151).

Zaidenberg, N. J., & Khen, E. (Zaidenberg et al 2015). Detecting Kernel Vulnerabilities During the Development Phase. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (pp. 224-230). IEEE

Zaidenberg, N., Neittaanmäki, P., Kiperberg, M., & Resh, A. (Zaidenberg et al 2015b). Trusted Computing and DRM. In *Cyber Security: Analytics, Technology and Automation* (pp. 205-212). Springer, Cham.

PIV

HYPERWALL: A HYPERVISOR FOR DETECTION AND PREVENTION OF MALICIOUS COMMUNICATION

by

Michael Kiperberg , Raz Ben Yehuda and Nezer Zaidenberg 2020

International Conference on Network and System Security. Best paper award

HyperWall: A Hypervisor for Detection and Prevention of Malicious Communication

Michael Kiperberg[1], Raz Ben Yehuda[2], and Nezer J. Zaidenberg[3]

[1] Software Engineering Department, Shamoon College of Engineering Beer-Sheva, Israel michaki1@sce.ac.il
[2] Department of Mathematical IT, University of Jyväskylä, Finland raziebe@gmail.com
[3] School of Computer Science, The College of Management, Academic Studies, Israel nzaidenberg@me.com

Abstract. Malicious programs vary widely in their functionality, from key-logging to disk encryption. However, most malicious programs communicate with their operators, thus revealing themselves to various security tools. The security tools incorporated within an operating system are vulnerable to attacks due to the large attack surface of the operating system kernel and modules. We present a kernel module that demonstrates how kernel-mode access can be used to bypass any security mechanism that is implemented in kernel-mode. External security tools, like firewalls, lack important information about the origin of the intercepted packets, thus their filtering policy is usually insufficient to prevent communication between the malicious program and its operator. We propose to use a thin hypervisor, which we call "HyperWall", to prevent malicious communication. The proposed system is effective against an attacker who has gained access to kernel-mode. Our performance evaluation shows that the system incurs insignificant (≈1.64% on average) performance degradation in real-world applications.

Keywords: Virtual Machine Monitors, Hypervisors, Trusted Computing Base, Network Security

1 INTRODUCTION

Malicious programs vary widely in their functionality, from key-logging to disk encryption. They utilize different vulnerabilities to achieve their goals: some attack applications [32] while others attack the kernel itself [17]. However, most malicious programs communicate with their operators, thus revealing themselves to various security tools. Specifically, firewalls attempt to detect and prevent such attacks by analyzing network packets that leave the network adapter. Unfortunately, firewalls must base their decision only on the content of the packets. The information about the origin of the packets, i.e., the name of the application that produced it, is lost. In practice, encryption is widely used for legitimate and malicious communication, thus firewalls have access only to the clear-text routing information, i.e., destination address and port. It is the responsibility of the

system administrator to configure the list of allowed or the list of restricted destinations. These lists can be constructed automatically using machine learning techniques [2, 13, 26, 8]. Similarly to traditional antiviruses, protection schemes based on black lists cannot withstand zero-day attacks, which are discovered weeks or months after infection [1].

On the other hand, software modules, like SELinux [31], can prevent the creation of sockets by unauthorized applications, regardless of their destination address. Unfortunately, these software modules, being part of the operating system, are vulnerable to attacks on the kernel and its modules [17].

Attacks on the kernel can be roughly divided into two categories: code attacks and data attacks. In code attacks [12], the goal of the attacker is to modify the instruction sequence. To achieve this goal, the attacker can replace the original code with their own or modify the control flow by manipulating the stack. Direct modification of the kernel code can be prevented using periodic verifications performed by the kernel itself, as done by Microsoft's Kernel Patch Protection [6], or by a more highly privileged software like a hypervisor [35].

Control flow integrity is a more challenging problem. It can be solved to some extent without recompilation of the kernel's code [22]. However better protection and performance can be achieved by analyzing the source code and recompiling the kernel [7].

Data attacks aim at altering the behavior of the kernel without modifying its code. The most common class of data attacks is Direct Kernel Object Modification (DKOM), in which kernel data structures are modified in order to achieve privilege escalation, process hiding [14], execution prevention [9], etc. Modern data attacks, called "data-oriented programming" achieve arbitrary computation (i.e., Turing complete) by assigning specially crafted values to the variables of a compromised program [11].

Due to the aforementioned security concerns, we see a migration of security modules from the kernel to an isolated environment. An isolated environment can be implemented using ARM's TrustZone [24], using a hypervisor that is available on most modern CPUs, or by introducing secure co-processor [21].

A hypervisor is a software component that has higher privileges than the operating system. Moreover, a hypervisor can configure interception of various events that occur in the operating system, thus making it an ideal candidate for security applications. For example, hypervisors can be used to protect the kernel code from modification [28], implement control flow integrity [22], provide additional security features like full disk encryption [29], etc.

The contribution of this paper is twofold. First, the paper describes a stealthy method by which malicious software can communicate with a remote operator. The communication cannot be detected by software modules executing inside the operating system because it is performed by direct manipulation of the network card registers. We demonstrate the viability of this method by implementing a kernel module for the Linux operating system. We note however that the same attack can be realized by employing random memory access vulnerability in the kernel. The method is not limited to the Linux operating system.

Then, the paper presents a design of a thin hypervisor we call "HyperWall" that detects and prevents malicious manipulation of network card registers. Because both legitimate and malicious accesses to the network card registers are performed from the kernel-mode, they are indistinguishable from the viewpoint of the hypervisor. Therefore, the hypervisor determines the legitimacy of access based on the content of the transmitted packets. Only packets that were previously transmitted by user-mode applications are considered legitimate.

Finally, we evaluate the security of the proposed solution and the impact of the hypervisor on the network and CPU performance. The results show insignificant (≈1.64% on average) performance degradation in real-world applications.

2 BACKGROUND

In 2005, Intel introduced an extension to their CPUs [20] that enables the execution of multiple operating systems simultaneously. Each operating system executes in an isolated environment, called a "Virtual Machine" (VM), and the executing of all the virtual machines is governed by an isolated software, called a "Virtual Machine Monitor" (VMM) or "hypervisor". The hypervisor can configure interception of various events that occur in the VMs, e.g., execution of privileged instructions, access to IO ports, triggering of an exception, access to Model-Specific Registers (MSRs), etc. When an event configured for interception occurs, the CPU transfers the control from the VM to the hypervisor. This transition is called a "VM-exit". The information about the occurred event is stored in a special data structure for the hypervisor's inspection.

The hypervisor reacts to a VM-exit by inspecting the information about the occurred event. Instructions that were configured to be intercepted trigger a VM-exit before their actual execution. Therefore the hypervisor must emulate these instructions after a VM-exit. During the emulation, the hypervisor advances the instruction pointer of the VM to the next instruction. After completion of the event handling, the hypervisor transfers the control to the VM by executing a special instruction (`VMRESUME`). This transition is called a "VM-entry".

In early versions of the virtualization extension, memory management had a high overhead due to frequent events that had to be intercepted, e.g., page table switching, page faults, page invalidations. In order to solve this problem, Intel introduced a secondary-level Address Translation (SLAT) mechanism, called "Extended Page Tables" (EPT) [10]. The hypervisor can define a page table for each VM, which defines translation of the VMs physical addresses to real physical addresses. Each entry of EPT defines not only the mapping between addresses but also the access rights. Using EPT, the hypervisor can isolate itself and the VMs from each other.

A hypervisor can intercept a wide range of events. We will discuss only three of them, which we use in this paper. The first type of event is access to MSRs. MSRs are used to report the features of a CPU and configure its state. They participate in the configuration of

– 64-bit environments via the `EFER` MSR,

– system call mechanism via the `STAR` family of MSRs and
– physical memory caching policies via the `MTRR` family of MSRs, etc.

Two special instructions, `WRMSR` and `RDMSR`, allow the software to write to and read from MSRs, which are identified by a number. The hypervisor can intercept read and write accesses to MSR separately of each MSR number and access type by setting an appropriate bit in the configuration of the VM (specifically in MSR bitmap field). Upon a VM-exit, the number of the MSR and its new value, in case of a write operation, are reported to the hypervisor.

Exceptions' delivery is another type of event. The hypervisor can intercept an exception before it is delivered to the VM by setting a bit that corresponds to the exception vector in the configuration of the VM (specifically in the exception bitmap field). Upon a VM-exit, the number of the exception as well as its error code is reported to the hypervisor.

The last type of event that has relevance to this paper is so-called "EPT-violations", i.e., access to the VMs physical addresses that cannot be translated to real physical addresses. In essence, EPT-violations are page faults in the secondary-level address translation. EPT-violations can occur either due to absent entries in the secondary level page table or due to inappropriate rights, e.g., write-access to a read-only page. Upon a VM-exit, the virtual and physical addresses are reported to the hypervisor. In addition the hypervisor receives a value that resembles an error-code. This value can be used to determine the reason for the EPT-violation.

2.1 Intel Network Cards

The attack that is presented in this paper targets the *i217* network card by Intel. The registers of this network card are memory-mapped, i.e., there is a set of physical addresses allocated for the network card, and values written to those addresses are transferred to the network card. Values read from those addresses are transferred from the network card. During its initialization, the operating system allocates two circular rings of descriptors `TX` and `RX`. The `RX` ring is used to transfer packets from the network card to the main memory. The `TX` ring is used to transfer packets from the main memory to the network card.

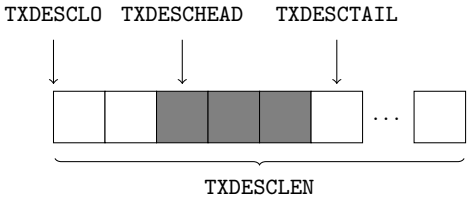


Fig. 1. Intel network card descriptor ring. The shaded descriptors are yet to be consumed by the network card.

Each ring is defined by four registers whose purpose is depicted in Figure 1. The `TXDESCLO` register holds the physical address of the base of the `TX` ring. The `TXDESCLEN` register holds the size of the `TX` ring. The `TXDESCHEAD` register holds the index of the first descriptor that is yet to be consumed by the network card. The `TXDESCTAIL` register holds the index of the first descriptor that is available for the operating system. Whenever possible, the network card consumes the descriptor indexed by `TXDESCHEAD` and advances the register to the next position. The network card stops when `TXDESCHEAD` reaches `TXDESCTAIL`.

Each descriptor points to a packet to be sent or provides some metadata (e.g., checksum offloading information) about the following packets. In order to send a packet, the operating system writes a descriptor to the `TX` ring at index `TXDESCTAIL` and advances the `TXDESCTAIL` register. In response, the network card consumes the descriptor and transmits the pointed packet or configures its internal state according to the metadata.

The network card provides checksum offloading functionality, i.e., it can automatically compute and insert checksums for the transmitted packets. Because the structures of headers vary between different protocols, before transmitting a serial of packets, the operating system prepares the network card for a specific protocol stack by inserting a metadata descriptor (or a context descriptor) with the relevant information. Metadata descriptors themselves do not transmit packets but rather affect the transmission of the following packets.

3 THREAT MODEL

For this paper, we assume that the attacker has random access to the kernel memory, i.e., he can read from and write to arbitrary locations in the memory by exploiting a vulnerability [17]. However, this assumption is restricted as follows:

- We assume that the operating system is equipped with a code integrity verification mechanism [28] and a control flow integrity verification mechanism [22], thus preventing direct code modification or control flow alteration.
- We assume that attacks on data buffers in user-mode applications are not feasible. To justify this assumption, we note that user-mode buffers, unlike kernel-mode buffers, can be swapped out and their location is less predictable.
- We assume that the system is equipped with a system call filtering mechanism, e.g., [23] that prevents unauthorized socket opening by the vulnerable application.
- We assume the existence of a peripheral firewall that blocks packets whose headers specify unsupported protocols, see Section 5.3.

To conclude, we assume that the attacker can read and write data (but not code) in kernel-mode (but not in user-mode).

4 ATTACK MODULE

We demonstrate the viability of a stealthy attack using a tiny kernel module. The module performs the required reads and writes. Although in reality, the attack would be performed from user-mode by exploiting an existing vulnerability in kernel-mode, we believe that the described kernel module is an adequate model for this attack as it does not use any functions or data structures of the kernel.

Strictly speaking, communication with a network card varies between different vendors and models. Some network cards can be configured to communicate via Memory-Mapped Input/Output (MMIO), whereas others require legacy input/output or even a combination of both. Only network cards that communicate via MMIO are vulnerable to the described attack.

The attack begins by obtaining the physical address of the MMIO region allocated by the firmware for the network card. This information can be obtained by enumerating the PCI configuration space or by asking the operating system (*Device Manager* on Windows and *lspci* on Linux).

The MMIO region maps the registers of a network card. Values written to those registers are transferred to the network card. Values read from those registers are transferred from the network card. The driver uses the four `TX`-registers to send a single UDP packet containing the string "hello".

The sending procedure can be divided into four steps:

1. locating the buffer that will store the UDP packet;
2. filling the buffer with a UDP packet;
3. writing the metadata and the regular descriptors; and
4. advancing the `TXDESCTAIL`.

In order to locate the buffer, the driver first reads `TXDESCLO` and `TXDESCLEN`. Then, the driver traverses the circular ring searching for a descriptor that points to a packet and that was already consumed by the network card. The buffer containing the consumed packet will be used as storage for the "hello" UDP packet. The driver fills the buffer with the "hello" UDP packet, including the UDP, IP, and MAC headers. Then, the driver writes two descriptors: a metadata descriptor that corresponds to a UDP over IP packet, and a regular descriptor which points to the buffer containing the "hello" UDP packet. Finally, the driver advances the `TXDESCTAIL` by two.

We note that all the steps performed by the driver require only memory reads and writes. Therefore, they could be performed from user-mode by exploiting random access vulnerability in kernel-mode.

5 HYPERVISOR DESIGN

5.1 Thin Hypervisors

HyperWall is a thin hypervisor that is capable of running only a single virtual machine. The hypervisor does not emulate any hardware devices and it allows

the operating system to execute normally without any interruption with a few exceptions that will be described in this section.

In security applications, thin hypervisors are preferable over a full hypervisor, like Xen, KVM [5], Hyper-V [33], etc., due to their lower performance impact and smaller attack surface. Xen 3.3, for example, consists of ≈320 KLOC [27] while HyperWall has only ≈3 KLOCs.

5.2 Hypervisor Initialization

HyperWall runs on Intel processors. The hypervisor is implemented as an application for the EFI firmware interface [38]. The EFI application is configured to load before the bootloader of the operating system. The EFI application loads a configuration file to be used by the hypervisor. The configuration file contains information about the offsets of kernel functions and variables that are used during the interception process. The offsets can be copied from the *System.map* file deployed with a kernel image. Figure 2 provides an example of a configuration file.

```
ffffffff81c00010 T entry_SYSCALL_64
ffffffff81908b00 T sock_sendmsg
ffffffff82158380 R inet_dgram_ops
ffffffff82158480 R inet_stream_ops
```

Fig. 2. Configuration file obtained from a *System.map* file

The installation procedure is simple. The EFI application containing the HV and the configuration file are copied to an EFI partition that contains the operating system or to a USB disk. Then, the boot order is changed to load the EFI application before the bootloader of the operating system.

5.3 Interception of System Calls

The hypervisor intercepts an execution of a single function `sock_sendmsg`, which is responsible for sending data though sockets. This function is invoked by the three system calls that send data through sockets: `sendto`, `sendmsg` and `sendmmsg`. Therefore by intercepting `sock_sendmsg`, we intercept all the data sending system calls.

The interception is performed by replacing the first instruction of the intercepted function with a software breakpoint instruction. This instruction triggers an exception on vector 3, which triggers a VM-exit. The hypervisor emulates the instruction that was replaced and advances the instruction pointer to the next instruction.

Due to KASLR [4], the virtual addresses of kernel functions change on each boot. In order to determine the actual virtual address, the hypervisor intercepts a write to the `LSTAR` MSR. The kernel writes to this MSR, the address of the system call handler, specifically the address of the `entry_SYSCALL_64` function. The difference between the actual address of the `entry_SYSCALL_64` function to the address that appears in the *System.map* file is the random relocation of the kernel. By adding this difference to any other symbol that appears in the *System.map* file we obtain its actual address.

The prototype of the function `sock_sendmsg`, which sends a message over a socket, is given in Figure 3. The function receives two parameters: `sock` and `msg`. The first parameter represents the socket over which the message is sent. The second parameter represents the message itself.

```
int sock_sendmsg(
  struct socket *sock,
  struct msghdr *msg)
```

Fig. 3. `sock_sendmsg` prototype

```
struct socket {
  ...
  const struct proto_ops *ops;
  ...
};
```

Fig. 4. `socket` structure

The first parameter points to a `socket` structure as shown in Figure 4. The hypervisor uses the `ops` field of this structure in order to determine the protocol of the socket. The `ops` field points to a set of functions that are responsible to handle actions performed on the socket. Each protocol has a separate set of handling functions. These sets are stored in global variables of the kernel. By comparing the `ops` fields to the values of these variables, the hypervisor can determine the protocol of the socket. HyperWall handles only two protocols and, therefore, uses only two variables: `inet_dgram_ops` for UDP and `inet_stream_ops` for TCP.

Communication protocols can be divided into two sets: C — carrier protocols, such as TCP and UDP, A — auxiliary protocols, such as ARP. The set of protocols that are handled by HyperWall should include all the protocols that

```
struct iovec {
  void __user *iov_base;
  __kernel_size_t iov_len;
};

struct iov_iter {
  ...
  const struct iovec *iov;
  unsigned long nr_segs;
  ...
};
struct msghdr {
  ...
  struct iov_iter msg_iter;
  ...
};
```

Fig. 5. `msghdr` structure

belong to C. Since protocols that belong to A cannot be used by an attacker, HyperWall does not handle these protocols. An external firewall should block all the packets belonging to C that are not handled by HyperWall.

The second parameter points to a `msghdr` structure as shown in Figure 5. The `msg_iter` field of this structure contains an array of buffers to be transmitted; specifically, the `nr_segs` fields represents the the number of buffers. The `iov` field is an array of buffers. Each buffer is described by an address (`iov_base`) and size ((iov_len)). The hypervisor computes a hash of each buffer and stores the hash in the hypervisor's internal data structure. The data structure is a balanced tree of a constant size. Initially, the tree is empty. When the tree reaches its maximal size, insertion of a new hash removes the eldest hash from the tree. This tree is used by the hypervisor for verification of packets that are transmitted by the network card. Because the operating system may attempt to retransmit a previously transmitted packet, it is incorrect to remove the hash upon a successful packet verification.

5.4 Interception of Network Card Accesses

The hypervisor uses an identity-mapping secondary-level page table. The hypervisor sets full access rights to all memory locations. There are only two exceptions for this setting. First, the region of physical memory containing the code and the data of the hypervisor is set to be inaccessible in the secondary level page table.

The second exception is the memory region containing the network card registers. The hypervisor configures the secondary-level page table such that any write attempt to the network card registers triggers a VM-exit. In response to

the VM-exit, the hypervisor emulates the instruction that accesses the registers and advances the instruction pointer to the next instruction. If the emulated instruction attempted to modify the `TXDESCTAIL` register, then the hypervisor performs a verification procedure prior to emulation, thus guaranteeing that non-authentic packets will be dropped.

The verification procedure consists of a loop in which the hypervisor verifies each packet that was transmitted by the operating system to the network card. More precisely, on an attempt to change the value of the `TXDESCTAIL` register from X to Y, the hypervisor verifies the packets pointed by descriptors at positions $X+1, X+2, \ldots, Y$ (circularly). The verification itself consists of two steps. First, the hypervisor computes the packet's hash. Then, the hypervisor checks whether the hash exists in the balanced tree. If not, then the hypervisor fills the corresponding descriptor with zeroes, thus preventing transmission of the malicious packet.

6 EVALUATION

6.1 Security Evaluation

We assess the security of the proposed system from two perspectives:

- We assess the ability of the hypervisor to protect its code and data.
- We assess the ability of the hypervisor to prevent malicious communication.

In order to protect its code and data, the hypervisor configures the secondary-level page table to prevent any access to its internal state. The hypervisor can also protect itself from DMA attacks by configuring the IOMMU page tables [18]. However, protection from DMA attacks was not implemented in our prototype. With an appropriate configuration, any attempt to access the physical pages containing hypervisor's internal state triggers an EPT-violation, allowing the hypervisor to respond. In our implementation, the hypervisor responds by entering an infinite loop.

HyperWall is implemented as an EFI application, which boots before the operating system. EFI firmware can verify the integrity of the EFI application via a feature called "Secure Boot" [38].

We assume that an attacker successfully compromised a user-mode application and the kernel. However, we assume that the compromised application does not have an open socket. We also assume that the attacker cannot manipulate the data sent to a socket that is open in another user-mode application. Therefore, the attacker can only manipulate kernel-mode buffers to achieve his goal. Examples of such buffers are *sk_buff*s, the address fields of the socket, and its state. By manipulating the buffer, the attacker can change the destination address and the content of a packet that was scheduled for transmission.

Alternatively, as described in Section 4, the attacker can write directly to the network card registers. This method is simpler because it does not depend on the existence of packets that were previously scheduled for transmissions. In

addition, this method does not require the attacker to know the exact layout of various kernel data structures. With the introduction of data structure layout randomization [3], this problem becomes particularly relevant.

The hypervisor compares the packet fetched by the network card to the packet submitted to `sock_sendmsg` function, and rejects malicious packets. The arguments of the `sock_sendmsg` function exist for a very short period of time and are pointed by local variables. Therefore, malicious modification of these arguments is unlikely.

We assume that the kernel is equipped with code and control flow integrity verification mechanisms. Therefore, the attacker will not be able to invoke the `sock_sendmsg` function directly and provide it with malicious arguments.

Our current implementation verifies only UDP and TCP packets; obviously, this can be extended to other protocols. Regardless of the set of implemented protocols, the peripheral firewall should be configured to block other protocols, thus preventing the attacker from using unverified protocols for communication. Auxiliary protocols, like ARP and ICMP should be allowed — but with care [30] — by the firewall.

6.2 Performance Evaluation

We evaluated the performance of HyperWall on a PC equipped with a 3.4GHz Intel Core i5-7500 CPU and 8GB of RAM running Ubuntu Desktop 18.04. We used the LM-Bench suite [19] for micro-benchmarking, and employed the Phoronix Test Suite (PTS) [15] to measure the performance impact on real-world applications.

To assess the performance impact of HyperWall on the kernel submodules, we used the LM-Bench. Table 1 compares latencies of various tests executed on a clean system and a system running HyperWall. The overhead of socket I/O is above 100% which can be explained by the additional interception of the involved system calls (which requires a context-switch) and hashing of the transmitted buffers.

To understand the performance of HyperWall in real-world applications, we used the Phoronix Test Suite. Table 2 compares the results of various tests executed on a clean system and a system running HyperWall. Different tests have different metrics: some measure latencies in seconds while others measure requests per second. The "Interpretation" column explains how the test metric should be interpreted in each case. The highest overhead is $\approx$7% while the average is $\approx$1.6%. In the SQLite test, the negative overhead is probably due to a measurement error.

7 RELATED WORK

The idea to use hypervisors in security applications is not new. In many applications, researchers extended Xen to provide additional security for the underlying VMs. SBCFI [22] is a Xen-based hypervisor that provides control flow integrity

Table 1. Performance results of vanilla Linux and HyperWall on the LMBench micro-benchmark.

Test Name	Vanilla Linux (μsec)	HyperWall (μsec)	Overhead
syscall	1.7996	1.8844	5%
read	0.4714	0.5892	25%
write	0.4340	0.5631	30%
fstat	0.4724	0.5838	24%
open/close	1.5462	2.4174	56%
select (10 fds)	0.5395	0.7116	32%
select (100 TCP fds)	5.2287	9.3992	80%
fork+exit	66.8446	80.2160	20%
fork+execve	210.5385	249.1556	18%
fork+/bin/sh	541.8947	1336.0000	147%
sigaction	0.4423	0.5148	16%
Signal delivery	1.0020	1.6766	67%
Protetcion fault	0.8230	1.1420	39%
Page fault	0.1495	0.3148	111%
Unix socket I/O	5.2694	7.8473	49%
TCP socket I/O	8.1917	18.4216	125%
UDP socket I/O	6.2157	16.4073	164%

Table 2. Performance results of vanilla Linux and HyperWall on the Phoronix Test Suite.

Test Name	Interpretation	Vanilla Linux	HyperWall	Overhead
SQLite	Lower is better	55.00 Seconds	53.64 Seconds	-2.47%
GnuPG	Lower is better	11.56 Seconds	11.58 Seconds	0.00%
PyBench	Lower is better	1.121 Seconds	1.198 Seconds	6.87%
Dbench	Higher is better	80.57 MB/s	79.80 MB/s	0.96%
IOzone	Higher is better	1224.79 MB/s	1193.18 MB/s	2.58%
PostMark	Higher is better	5813 Req/s	5813 Req/s	0.00%
PHPBench	Higher is better	610,335 (Score)	588,940 (Score)	3.51%

verification for the underlying operating system. Because HyperWall depends on CFI mechanism for the kernel, SBCFI can complement our hypervisor. Nitro [23] is a Xen-based hypervisor for system call tracing. IntroVirt [37] is a stealthy Xen-based hypervisor that hooks system calls in Windows for introspection. We use a similar idea for hooking an inner function that is invoked by several system calls. Another Xen-based hypervisor for ARM is described in [25]. The hypervisor is capable of performing stealthy instrumentation thus allowing for dynamic malware analysis. The mechanism of stealthy instrumentation can be introduced into HyperWall to make it suitable for malware analysis.

Thin hypervisors are not as prevalent as full hypervisors, probably due to the additional development effort compared with using an existing open-source full hypervisor. SecVisor [28] is a thin hypervisor that uses a secondary level address translation and IOMMU to prevent unauthorized code execution in kernel-mode. Functionality similar to SecVisor's must be added to our hypervisor to guarantee the kernel's code integrity. BitVisor [29] is a thin hypervisor that intercepts accesses to ATA hard disks and enforces storage encryption. HyperWall uses a similar method of IO interception. HyperSafe [34] is a CFI extension to BitVisor.

Secloack [16] is a security component that is able to reliably turn peripheral devices of a smartphone on and off. The component resides in ARM's TrustZone. Secloack prevents communication with peripheral devices completely, whereas HyperWall performs filtering on this communication.

HookMap [36] is a rootkit detector that is based on QEMU. Similarly to HyperWall, it uses the *System.map* file to obtain the locations of kernel functions.

8 CONCLUSIONS

Although it seems to be a difficult task to prevent the execution of malicious programs, it may be possible to detect and block their malicious behavior. In this paper, we presented a system that can detect and prevent malicious communication, which is essential for the operation of malicious programs. The system can withstand attacks from kernel-mode and incurs low overhead in real-world applications.

References

1. Bilge, L., Dumitraş, T.: Before we knew it: an empirical study of zero-day attacks in the real world. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 833–844 (2012)
2. Bilge, L., Sen, S., Balzarotti, D., Kirda, E., Kruegel, C.: Exposure: A passive dns analysis service to detect and report malicious domains. ACM Transactions on Information and System Security (TISSEC) **16**(4), 1–28 (2014)
3. Chen, P., Xu, J., Lin, Z., Xu, D., Mao, B., Liu, P.: A practical approach for adaptive data structure layout randomization. In: European Symposium on Research in Computer Security. pp. 69–89. Springer (2015)
4. Cook, K.: Kernel address space layout randomization. Linux Security Summit (2013)

5. Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., Rao, B.: Quantitative comparison of Xen and KVM. Xen Summit, Boston, MA, USA pp. 1–2 (2008)
6. Ermolov, M., Shishkin, A.: Microsoft windows 8.1 kernel patch protection analysis
7. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 179–194. IEEE (2016)
8. Ghafir, I., Prenosil, V.: Dns traffic analysis for malicious domains detection. In: 2015 2nd International Conference on Signal Processing and Integrated Networks (SPIN). pp. 613–918. IEEE (2015)
9. Graziano, M., Flore, L., Lanzi, A., Balzarotti, D.: Subverting operating system properties through evolutionary dkom attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer (2016)
10. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System programming Guide, Part **2**, 11 (2011)
11. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 969–986. IEEE (2016)
12. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: USENIX security symposium. pp. 383–398 (2009)
13. Kheir, N., Tran, F., Caron, P., Deschamps, N.: Mentor: positive dns reputation to skim-off benign domains in botnet c&c blacklists. In: IFIP International Information Security Conference. pp. 1–14. Springer (2014)
14. Korkin, I.: Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel. arXiv preprint arXiv:1805.11847 (2018)
15. Larabel, M., Tippett, M.: Phoronix test suite. Phoronix Media, [Online]. Available: http://www.phoronix-test-suite. com/.[Accessed June 2020] (2020)
16. Lentz, M., Sen, R., Druschel, P., Bhattacharjee, B.: Secloak: Arm trustzone-based mobile peripheral control. In: Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services. pp. 1–13 (2018)
17. Lu, S., Lin, Z., Zhang, M.: Kernel vulnerability analysis: A survey. In: 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC). pp. 549–554. IEEE (2019)
18. Markuze, A., Morrison, A., Tsafrir, D.: True iommu protection from dma attacks: When copy is faster than zero copy. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 249–262 (2016)
19. McVoy, L.W., Staelin, C., et al.: lmbench: Portable tools for performance analysis. In: USENIX annual technical conference. pp. 279–294. San Diego, CA, USA (1996)
20. Neiger, G., Santoni, A., Leung, F., Rodgers, D., Uhlig, R.: Intel virtualization technology: Hardware support for efficient processor virtualization. Intel Technology Journal **10**(3) (2006)
21. Petroni Jr, N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot-a coprocessor-based kernel runtime integrity monitor. In: USENIX security symposium. pp. 179–194. San Diego, USA (2004)
22. Petroni Jr, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 103–115 (2007)

23. Pfoh, J., Schneider, C., Eckert, C.: Nitro: Hardware-based system call tracing for virtual machines. In: International Workshop on Security. pp. 96–112. Springer (2011)
24. Pinto, S., Santos, N.: Demystifying ARM TrustZone: A comprehensive survey. ACM Computing Surveys (CSUR) **51**(6), 1–36 (2019)
25. Proskurin, S., Lengyel, T., Momeu, M., Eckert, C., Zarras, A.: Hiding in the shadows: Empowering arm for stealthy virtual machine introspection. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 407–417 (2018)
26. Rahbarinia, B., Perdisci, R., Antonakakis, M.: Segugio: Efficient behavior-based tracking of malware-control domains in large isp networks. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 403–414. IEEE (2015)
27. Rutkowska, J., Wojtczuk, R.: Preventing and detecting xen hypervisor subversions. Blackhat Briefings USA (2008)
28. Seshadri, A., Luk, M., Qu, N., Perrig, A.: 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. pp. 335–350 (2007)
29. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., et al.: Bitvisor: a thin hypervisor for enforcing i/o device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 121–130 (2009)
30. Singh, A., Nordström, O., Lu, C., Dos Santos, A.L.: Malicious icmp tunneling: Defense against the vulnerability. In: Australasian Conference on Information Security and Privacy. pp. 226–236. Springer (2003)
31. Smalley, S., Vance, C., Salamon, W.: Implementing selinux as a linux security module. NAI Labs Report **1**(43), 139 (2001)
32. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy. pp. 48–62. IEEE (2013)
33. Velte, A., Velte, T.: Microsoft virtualization with Hyper-V. McGraw-Hill, Inc. (2009)
34. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE Symposium on Security and Privacy. pp. 380–395. IEEE (2010)
35. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 545–554 (2009)
36. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: International Workshop on Recent Advances in Intrusion Detection. pp. 21–38. Springer (2008)
37. White, J.S., Pape, S.R., Meily, A.T., Gloo, R.M.: Dynamic malware analysis using introvirt: a modified hypervisor-based system. In: Cyber Sensing 2013. vol. 8757, p. 87570D. International Society for Optics and Photonics (2013)
38. Wilkins, R., Richardson, B.: Uefi secure boot in modern computer security solutions. In: UEFI Forum (2013)

PV

PROTECTION AGAINST REVERSE ENGINEERING IN ARM

by

Raz Ben Yehuda, Nezer Zaidenberg 2011

Proceedings of Industrial Technology (ICIT), IEEE International Conference on.
2011

International Journal of Information Security Special Issue on "IoT Security and Privacy" manuscript No.
(IJIS-D-18-00538R2)

Protection against reverse engineering in ARM

Raz Ben Yehuda[a,1,3], **Nezer Jacob Zaidenberg**[b,1,2,3]

[1]University of Jyväskylä, Jyväskylä, Finland
[2]College of Management Academic Studies, Street, Rishon LeZion, Israel
[3]TrulyProtect, Finland

Received: 09 Nov 2018/ Accepted: 18 Jun 2019

Abstract With the advent of the mobile industry, we face new security challenges. ARM architecture is deployed in most mobile phones, homeland security, IoT, autonomous cars and other industries, providing a hypervisor API (via virtualization extension technology). To research the applicability of this virtualization technology for security in this platform is an interesting endeavor. The hypervisor API is an addition available for some ARMv7-a and is available with any ARMv8-a processor. Some ARM platforms also offer TrustZone, which is a separate exception level designed for trusted computing. However, TrustZone may not be available to engineers as some vendors lock it. We present a method of applying a thin hypervisor technology as a generic security solution for the most common operating system on the ARM architecture. Furthermore, we discuss implementation alternatives and differences, especially in comparison with the Intel architecture and hypervisor with TrustZone approaches. We provide performance benchmarks for using hypervisors for reverse engineering protection.

Keywords Security · ARM · Mobile · IoT · Hypervisor

1 Introduction

We explore Man-at-the-Endpoint (MATE) attacks on ARM-based systems such as embedded and mobile devices and focus on protection against reverse engineering for the ARM platform.

The TrulyProtect [3] microrvisor is a secured thin hypervisor based on the blue pill concept [8] for the x86 environment. TrulyProtect does not run on multiple operating systems and is used only for security purposes. TrulyProtect was initially implemented and benchmarked on an x86 CPU architecture. We examine the necessary modifications for porting TrulyProtect to ARM, its security benefits, and performance costs. TrulyProtect provides a thin layer of code that is invoked through traps on the x86 architecture. We use a similar approach using exceptions to elevate from lower privilege levels in ARM. These traps can be generated from userspace programs (EL0) or kernel code (EL1).
In general, our technique is composed of the following steps:

[a]e-mail: raz@trulyprotect.com
[b]e-mail: nezer@trulyprotect.com

- Static phase
  Encrypt some segment of the ELF binary.
  Replace this segment with a trap opcode.
- Runtime phase
  Whenever the processor executes the trap opcode, the processor moves from user mode to HYP mode, decrypts the encrypted code, executes it in HYP mode, encrypts it back and returns to user mode.

In this paper, we examine the anti-reverse engineering of native code. We present a thin hypervisor implemented in an ARMv8-a 64-bit processor, and synthetic benchmarks. The hypervisor can be considered a Trusted Execution Environment (TEE) as it offers an isolated execution environment for decrypted code.

2 Background

We present the ARM architecture permission model and past work on TrulyProtect under an Intel platform.

2.1 ARM permission model

The ARMv8-a platform normally has four exception (permission) levels:

Exception level 0 (EL0) refers to normal userspace code. EL0 is analogous to "ring 3" in the x86 platform. For example, virtually all applications and games on a standard iPhone or Android phone run on EL0.

Exception level 1 (EL1) refers to operating system code. EL1 is analogous to "ring 0" in the x86 platform. For example, the Android or the iOS (the operating system itself) on a mobile phone runs on EL1.

Exception level 2 (EL2) refers to HYP mode. EL2 is analogous to "ring -1" or "hypervisor mode" on the x86 platform. In most ARM devices, nothing runs on this exception level unless the ARM device starts a hypervisor when it boots.

Exception level 3 (EL3) refers to TrustZone™.
TrustZone is a special security mode that can monitor the ARM CPU as well as the operating system that it runs. TrustZone allows running a separate security real-time operating system in a secure world. There are no directly analogous modes, but similar concepts in x86 (from security perspectives) are Intel's ME [9] and SMM [10].
Each of the exception levels provides its own state of registers and can access the registers that correspond to the lower permission levels, but not the higher levels. In Figure 1, Exception Levels EL3, EL2 and EL0/EL1 have their own translation tables. Thus, moving between exception levels requires a change of the entire address space.

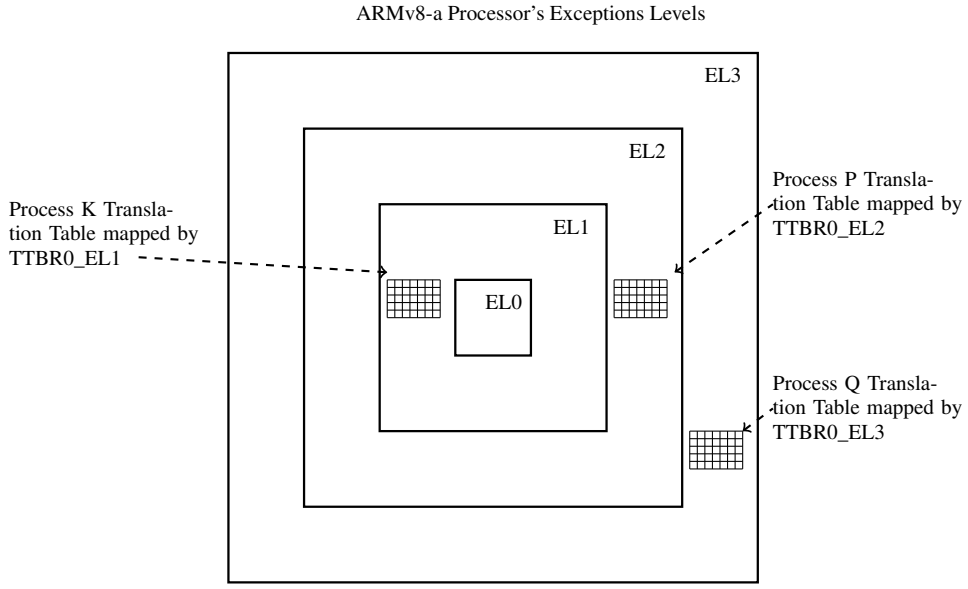


Fig. 1 MMUs separation

2.2 TrustZone

EL3 refers to the TrustZone architecture [11, 12]. TrustZone is a special ARM security mode that allows the running of a "secure OS" in parallel to the normal "insecure OS." The normal (insecure OS), called "normal world," is the standard operating system that normally runs on the device (for example, VxWorks, Linux, iOS or Android). The secure OS, called "secure world," is an implementation-specific real-time operating system (such as OKL4[13]) and is chosen by the hardware vendor. The purpose of the secure world is to attest the normal world and provide a root of trust and trusted computing services. The secure world is separated from the non-secure world by the hardware memory protection unit.

Users cannot normally run applications in the secure world. Furthermore, running applications on TrustZone is usually prevented by the vendors.

ARM Holdings introduced TrustZone as part of its architecture in 2009. TrustZone was an optional extension to the ARMv7-a architecture. TrustZone is part of the ARMv8-a architecture and described in detail by Ngabonziza [14]. The TrustZone can be used in two distinct operational modes:

1. Monitor mode provides a separate operating system to run concurrently with a second, generic operating system (such as Linux). The ARM processor itself assigns resources (such as processing cycles) to the TrustZone real-time operating system periodically.
2. Passive library mode includes a monitor exception vector that is activated through traps or SMC calls.

It might have been possible to implement TrustZone as a virtual machine running on EL2. However, TrustZone is designed to execute in an isolated environment and at a higher privilege level than the hosting machine and hypervisor.

Unfortunately, a malicious application running in TrustZone or HYP mode may be used to attack the TrustZone operating system or the hypervisor itself. However, these attacks

are only possible if the malicious application runs in EL2 or EL3. Fortunately, installing malicious input can be detected by TrustZone in the Static Root of Trust Measurements (SRTM) process.

2.3 TrulyProtect

On x86 platforms TrulyProtect provides anti-reverse engineering [15], endpoint security [16], video decoding [25], forensics etc. TrulyProtect relies on Dynamic Root of Trust Measurement (DRTM) attestation to create a trusted environment in the hypervisor and receive encryption keys [17].

After receiving the keys, TrulyProtect provides anti-reverse engineering by executing encrypted code in the hypervisor and protecting the decryption keys. In this work, we review the performance and capability of similar hypervisor operations on the ARM architecture. Additional work includes endpoint security by controlling instructions (memory pages) that are allowed on an endpoint. The hypervisor reads an encrypted and signed database of allowed pages. The hypervisor grants execution permission only to pages that are allowed to execute on the endpoint. Platform independent extensions to the TrulyProtect hypervisor also allow protection against reverse engineering of managed code and protection of encrypted video. These features could be ported to the ARM architecture as well but were beyond our scope.

2.4 Attestation

TrulyProtect DRTM (Dynamic Root of Trust Measurements) relies on an attestation method based on a technique similar to Kennel and Jamieson [27,28]. This attestation method is not required on ARM (though we believe similar methods can be developed); thus, the Static Root of Trust Measurements (SRTM) using TrustZone is used instead. If TrustZone is unavailable, using TPM [19] and Secureboot is another attestation option [15]. If the hardware root of trust is not available, then even DRTM [29] is possible. Regardless, we provide the decryption keys only after successful attestation. We have no innovation in this field and thus, attestation remains out-of-scope.

2.5 Protecting the hypervisor

In embedded devices, such as mobile phones or ARM servers used for cloud computing, a malicious user may try to compromise the computer's [30] hypervisor [31,32] as also described in [33] for Xen or KVM [35] through a host privilege escalation. Min [34] claims that the best approach is to rely on the hardware and presents a security monitor as a solution. We agree with this approach and offer to do the attestation in, for instance, the TrustZone. However, we examine what possible vulnerabilities a malicious software might try to take advantage of:

- A boot loader or kernel replaced. The chain of trust would prevent the bootloader or the kernel from executing.
- A malicious program. A TrulyProtect's signed program with a malicious code generates the escape sequence to enter the VM. This means that part of the code tries to access privileged registers. We trap any access to these registers while executing in EL2 so any

attempt would be trapped to the hypervisor. For example, we disable the SMC calls and HVC calls while executing in EL2. In cases when there is no protection in EL2, we can protect from the EL3 (TrustZone).
– A malicious hypervisor. It is not possible to replace the TrulyProtect hypervisor once it is set.

Once a trustworthy hypervisor is running, it can protect the keys.

3 Related works

Many researched techniques for trusted execution, in this section, we describe some.

3.1 TEE in ARM

Ekberg et al [43] discuss TEE in mobile devices as having strict requirements that date back to the beginning of the mobile device industry and describes the various techniques of protection, a chain of trust, trusted storage, and others. They also mention virtualization as a means of protection through the use of a VMM (Virtual Machine Monitor). These VMMs run several concurrent guests isolated one from the other. The TrulyProtect thin hypervisor does not rely on a VMM and is not considered a guest.

3.2 Code obfuscation and DRM

A program can be obfuscated to diminish the chances of successful reverse engineering and discovery of its trade secrets, modification, etc. There has been significant work on systems to obfuscate and de-obfuscate code. Vot4CS [46] is a relatively recent obfuscation for C# that survives many de-obfuscation attempts. Kevin Coogan et al [7] and Kalysch [42] describes means to attack such obfuscators. Szor [36] discusses automatic de-obfuscation in order to detect computer viruses. Such methods allow code to execute on a target machine and make reverse engineering much more challenging. However, code obfuscation is bound to fail eventually [47]. In practice, experience shows that by investing time and dedication, these methods can frequently be broken faster than people think. For example, the Nintendo Wii-U DRM system is notorious for being broken less than one month after the platform was released despite Nintendo having full control on the hardware operating system and the software [38].

DRM [17] techniques require protection against reversing as they are a frequent target of reversing and removal attempts. TEE and reverse engineering technologies can be used on many entertainment systems for DRM protection. Devices such as smart TVs, handheld devices, TV set-top boxes and game consoles [6] are all examples of using modern hardware for these purposes. Hardware modern security features are used to prevent copying whenever such features are available.

3.3 Intel SGX

Intel SGX [1] is a set of instructions added to the processor that enables the use of protected and isolated memory regions knows as "enclaves." Access to such enclaves requires special software tools and expertise.

3.4 OKL4 Microvisor

OKL4 Microvisor [18] is a secure hypervisor supported by Cog Systems. The OKL4 microvisor supports both para-virtualization and pure virtualization. It is designed for the IoT industry, and supports ARMv6, ARMv5, ARMv7-ve [4] and ARMv8-a [5]. Unlike TrulyProtect, the OKL4 microvisor is a full kernel executing in HYP mode. Cog Systems also offers an SDK called "D4 secure SDK" and an RTOS.

4 Anti-reverse engineering

TrulyProtect [3] for the ARM thin hypervisor offers an easy way to execute code in a secure environment in ARMv8-a, and does so in a way that does not require the user to modify the code. Unlike QSEE [20,21], the interaction with the secured area does not require any special preparations. TrulyProtect is a real thin hypervisor, i.e., its footprint is less than 100kB when counting the AES decryption. It does not offer a system-wide solution but focuses on protecting distinct parts of the program. Memory protection, anti-reverse engineering and protection of keys are all implemented like other platforms supported by TrulyProtect.

5 Innovation

In this section, we detail the proposed system for anti-reverse engineering in ARM.

5.1 Program protection in ARM

We now examine the anti-reverse engineering process. Anti-reverse engineering is often achieved using obfuscation; however, here, we want to prevent the reverse engineering of software by encrypting the software code before deployment and deploying only the encrypted software. We make the following assumptions:

- The encryption function we use is safe and cannot be broken. We use AES [22]; Nevertheless, if in the future, AES is broken [23] then the encryption function can easily be replaced with an elliptic curve [24] or any other encryption function. (We do not assume anything about the encryption function.)
- The CPU itself is sufficiently complex to prevent the attacker from "looking" inside the CPU.
- We assume the CPU works according to the specifications and no hidden modes allow internal CPU structures to be read.

5.2 The proposed system

Under the above assumptions, we provide evidence that the decrypted software is not available to the normal operating system, and the hypervisor will undertake the protection for the software and the decryption keys. This proposed system is composed of two phases.

1. Static encryption
   We choose which functions we wish to encrypt and obtain a binary copy of the program. Then we use TrulyProtect's instrumentation tool to encrypt the chosen functions.

2. Runtime execution
   The program runs as-is. Each time the processes access the encrypted function, the processor drops to HYP mode, decrypts the function, and executes it.

5.2.1 Static encryption

As noted earlier, a program protection framework in ARM mimics the way TrulyProtect protects programs in x86. However, because the ARM hypervisor differs greatly from other hypervisors (such as the x86 hypervisor), we will describe how it is being undertaken in detail.
Figure 2 depicts the first stage of the encryption of a single function. An ELF ARM binary is processed, and the instructions of the function foo() are replaced with a trap code, the "BRK" instruction.

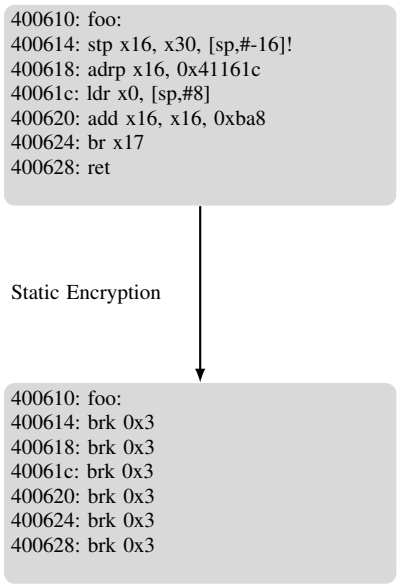


Fig. 2 Static Encryption - replace a function

We chose the "BRK" instruction for two reasons:

1. It easy to configure to trap into the HYP mode by setting the mdcr_el2 register.
2. It does not change any value of the general purpose registers; thus, when trapping to the hypervisor, the program's context can be saved.

The addresses from the left are relocatable, meaning that the actual addresses are not known at the static encryption phase. It is important to note that the ARMv8-a exception level model dictates that only positive addresses, i.e., userspace addresses can be executed in EL2, and not negative (kernel addresses).
Static encryption also includes the addition of the new encrypted function foo(). The encrypted version of the function is added to the ELF binary as a new segment, as depicted in Figure 3.

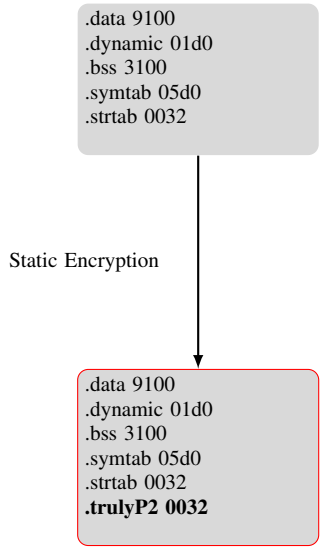


Fig. 3 Static Encryption - additional segment

5.2.2 Runtime decryption

The next phase is when the program is loaded and executed. To understand this phase we need to explain ARM's memory model. But first, we start with the x86 memory model. In x86 accessing a process or a kernel's virtual memory from HYP mode does not require mapping (Figure 4).

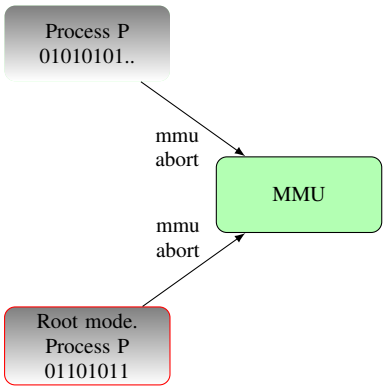


Fig. 4 x86 model. A single translation table

However, in ARM it is required to map the designated pages to the hypervisor translation table. For this reason, when we want to access EL0 code or data from EL2, we must first map the pages to the hypervisor. As a result, a page is mapped twice: one to each of two distinct memory tables of the same process, as shown in Figure 5.

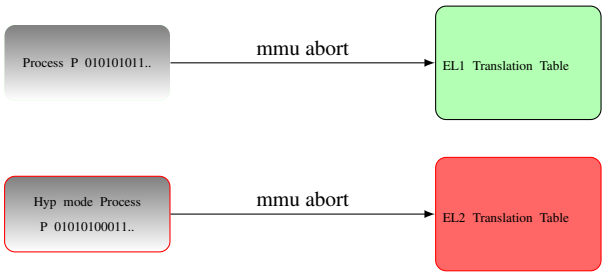
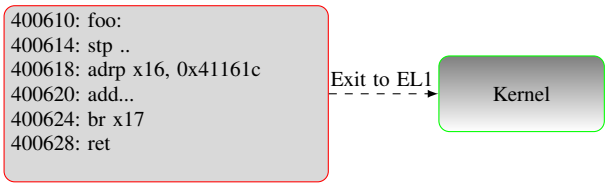


Fig. 5 ARM model. Translation tables are not shared

Thus, when any arbitrary program executes, the kernel searches in the ELF segments for the segment .trulyP2, and if it is found, then the kernel calls the hypervisor to enable trapping the "BRK" instruction. As long this process runs, any time the processor executes "BRK", it will trap into HYP mode. The hypervisor verifies that the current position of the instruction pointer is in foo(), and if so, it decrypts the encrypted version of foo() accommodated in .trulyP2 onto its original position. Then it starts executing from where the trap was and continues to execute in EL2 until one of the followings occurs:

1. foo() reaches its end, performs the ret instruction, and returns to the hypervisor. The hypervisor flushes the caches and TLBs, put back the trap code and return to EL0.
2. foo() accesses unmapped memory which results in an EL2 MMU abort (Figure 6.)
3. foo() performs an svc (a system call).

The ARM memory model poses a problem in MMU aborts. If there is an MMU data abort in EL2, then the unmapped region must also be mapped to EL0 memory page tables if it was not already mapped before. For this reason, we only map the trap-code (the code that generates the exceptions), the encrypted code, and the stack. This way, we know that the MMU aborts because of an unmapped region in EL2 while the region is mapped in EL1/EL0 (or will be mapped). As we show later, this approach has a severe performance penalty. For this reason, we tried a different approach - a real-time mapping, referred to here as **Rt-map**.

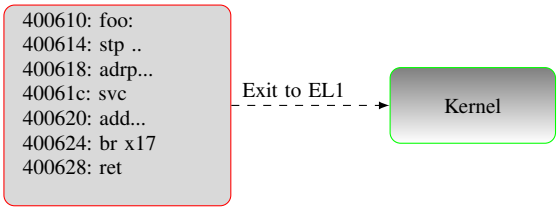


Instruction 400618 generated an MMU abort because address 0x41161c was not mapped to EL2

Fig. 6 Real-time mapping in EL2

In Rt-map mode, when an MMU abort takes place in EL2, we exit to EL1 and access the faulting address to map it to EL0/EL1 (assuming it was not already mapped); then, we map

the faulted address to EL2 and continue execution in EL0. With this approach, the next time this code is accessed in EL2, it will not MMU-abort in EL2 as previously but will continue to execute until the next MMU abort. This way, most of the process's address space gradually maps to EL2 as the process executes. Consequently, the process exits to EL0 (and then EL1) only when it performs a system call (Figure 7). Thus, an interposition is kept between the EL2 and EL0 processes.



The SVC is designed to exit from EL2 to EL0. The program counter is programmed to re-execute the SVC in EL0 again so that the SVC trap would be generated from EL0 to EL1

Fig. 7 System call in EL2

This way, the decrypted content is never available to the operating system. If a user tries to dump the memory image of the process while it is in EL2, then the processor would leave HYP mode and, consequently, put back the trap-code.

5.3 The hyplet

Protecting EL1

The hyplet is a generic term we use to describe programs that partly execute in EL0 or EL1 and partly in EL2. The technique we introduce in this paper is a special case of an encrypted hyplet. The hyplet as a general term is not part of the paper, but we provide some details for how it works. It is a challenging task to protect device drivers in ARM8v-a. For this reason, we developed the hyplet ISR (Interrupt Service Routine), in short hypISR. The hypISR is a means to move from EL1 to the userspace program without a noticeable penalty. A hyplet-ed program is a program that constantly accommodates EL2 translation table, and is not evacuated from EL2's MMU.

Whenever an interrupt is being processed by the processor; if the data or code need to be protected, then the processor moves to EL2 and from there to the userspace program that handles it. The callback function that processes this interrupt routine is the hyplet. Through this dedicated userspace process, sensitive data can be protected by reading it into a protected memory region.

To ensure that both the program code and data are always accessible, it is essential to disable evacuation of the program's translation table from the processor. Therefore, we chose to constantly accommodate the code and data in the hypervisor translation registers (Figure 8) [4]. To map a userspace program, we modified the Linux ARM-KVM mapping infrastructure to map userspace code with kernel-space data [37].

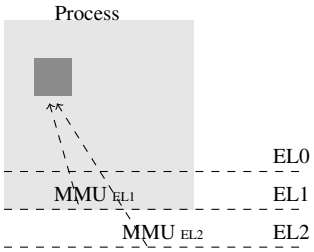


Fig. 8 Asymmetric dual view

Figure 8 shows identical addresses mapped to different virtual addresses in two separate exception levels. The small square subsection is mapped to EL2 and is therefore accessible from EL2. Usually, when executing in EL2, EL1 data is not accessible without premature mapping to EL2. This mapping extends the ability of a Linux process to execute from two exception levels to three.

Protecting the hypervisor MMU

In the standard method for memory mapping, EL1 is responsible for EL1/EL0 memory tables, and EL2 is responsible for its own memory table, in the sense that each privileged exception level accesses and manages its own memory tables (Figure 9).

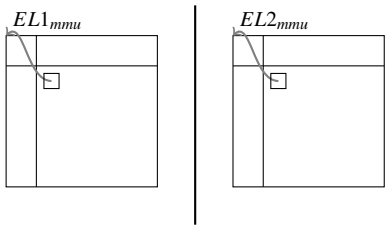


Fig. 9 Memory table access

This approach, however, puts the microvisor at risk because it might overwrite or otherwise garble its own page tables. As noted earlier, ARM8v-a hypervisor has a single memory address space (unlike TrustZone that has two, for the kernel and the userspace). The ARM architecture does not coerce an exception level to control and access its own memory tables, allowing the ARM architecture to map the EL2 page table in EL1 (Figure 10). As such, only EL1 can manipulate the microvisor page tables.

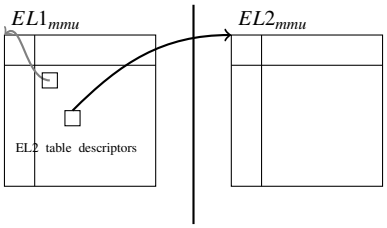


Fig. 10 Memory table access for hyplets

5.4 OS dependence

The ARM hypervisor is designed to be distinct in execution and memory context from other exception levels. As a result, whenever the hypervisor needs to access EL0 pages, it must map them to its own translation table. However, for that to happen, the hypervisor must implement its own page allocation system because there might be a need to allocate a table. However, we refrain from accessing the translation tables of EL2 in EL2 to reduce risks. Also, it is better to re-use the KVM-ARM [37] allocation system because it is a mature software. For these reasons, we decided to use KVM-ARM; thus, the hypervisor in this paper refers only to Linux implementations.

6 Evaluation

In the following sections, we estimate our microvisor overhead. we evaluate IPA overhead and TrulyProtect technology to TrustZone based solutions and measure encrypted program execution overhead. We measure the penalty of the repeated encryption, cache and TLB evacuations that are caused by system calls or minor page faults in EL2. We test the microvisor in typical I/O operations and under CPU loads. We also describe the means to mitigate these penalties.
The measures presented in the tests are averages, standard deviations, minimum and maximum. Each experiment was performed five times.
The tests were conducted in a Lenovator “Hikey” board. A Hikey is a small system-on-a-chip ARM-based computer manufactured by LeMaker. Hikey’s processor is an ARMv8-a.

SoC	HiSilicon Kirin 620
Number of CORES	8
Frequency	1.2 GHz
RAM	2GB
RAP-TYPE	LPDDR3 1.6 GHz

Table 1 Test Hardware

The software used was:

Linux Kernel Version	4.4.11
Distribution	Debian
Compiler	gcc-linaro-4.9-2015.02

Table 2 Test Software

6.1 IPA overhead

We wish to evaluate the cost of using the Intermediate Physical Address (IPA) [39]. The IPA is a second stage of translation and is used to separate the guest operating system from the physical memory by a double memory fault mechanism.

We measured the overhead of a two-stage translation compared with a single stage translation. The test software was RAMspeed.

The tests were conducted by two kernels with the same configuration.

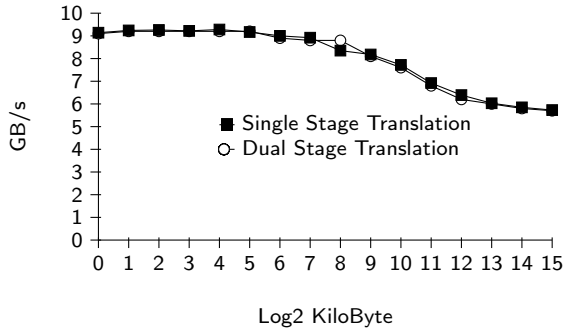


Fig. 11 IPA vs Native, WRITE access

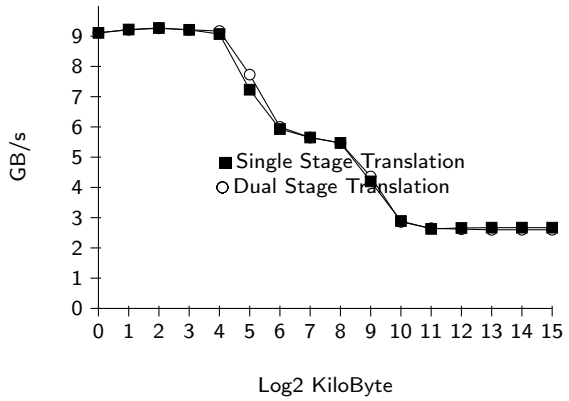


Fig. 12 IPA vs Native, READ access

In Figure 4 we attempt to simulate more closely the real-world computing load. A, B, and C are locations in the memory; M in SCALE is a constant.

SCALE	A = m*B
ADD	A + B = C
COPY	A=B

Table 3 IPA Tests explained

Single stage	COPY	SCALE	ADD
Avg	2.03	1.60	1.65
StdDev	0.01	.0.1	0.008
Max	2.05	1.62	1.66
Min	2.02	1.59	1.64

Dual stage	COPY	SCALE	ADD
Avg	2.08	1.62	1.69
StdDev	0.01	0.005	0.008
Max	2.1	1.63	1.7
Min	2.08	1.62	1.68

Table 4 Real Load GB/s

Evidently, there is no difference between using a two-stage translation and a single stage translation. We have shown that IPA does not influence the memory access performance.

6.2 TEE enter/exit overhead

The common way to enter TrustZone is when the processor executes the SMC call from EL1. In TrulyProtect, the processor traps to EL2 when it executes the "BRK" instruction. TrustZone implementations include a kernel space driver and the data are passed through ioctl(2)s operations that use SMC. In this test, we evaluate how costly the "BRK" trap is, when it traps to EL2, compared with the SVC/HVC (SVC is a system call to EL1 and HVC is a hypervisor call to EL2) and ioctl (2) [41] system call. We assume the HVC overhead is similar to SMC.

Trap	BRK trap	Pure SVC call	Ioctl syscall
Avg	92 ns	92 ns	490 ns
StdDev	41 ns	41 ns	65 ns
Max	156 ns	156 ns	550 ns
Min	52 ns	52 ns	440 ns

Table 5 BRK vs. SVC & Ioctl

There is a very little difference when comparing a trap to a typical SVC call. Executing ioctl that does nothing costs on average about 500ns without pre-emption. This is because there are many operations undertaken by the kernel before it returns. Since executing a

"BRK" trap takes on average 100ns and an ioctl takes 500 ns, we can say that the TrulyProtect mechanism for entering and exiting the TEE is five times faster than TrustZone-based technologies.

6.3 Encrypted code CPU use

We have used an FFT (Fast Fourier Transformation) program as a benchmark. The FFT program is a CPU-intensive program. This program does not perform any IO-related tasks. We chose this program because it traps to EL2 when it first enters the encrypted function and then continues to run until it reaches its end, and returns to TrulyProtect's hypervisor. There are no traps from EL2 to EL2 in this test. The following presents several decryption strategies.

- **Native** A clear-text run.
- **Live-decrypt** Decrypts in real time and flushes the instruction cache on exit from HYP mode. When the hypervisor is entered again, it decrypts again.
- **Cache** The first time the hypervisor enters the encrypted code, it caches the decrypted code into a temporary protected buffer and from now on the buffer is copied onto the trap-code each time the hypervisor is entered for execution. On exit, the function's decrypted instructions are flushed from the L1 instruction cache. (non-transient write-back).
- **RT-map** In addition to caching the decrypted data, i.e.; the "cache" mode, the hypervisor maps other parts of the process's address space into the hypervisor in real time to reduce exits from HYP mode. Reducing the exits reduces the cache flushes and copying.

6.3.1 First execution overhead

We measured the duration of an encrypted function when it first entered. The overhead includes the context switch to hypervisor mode and the time required for the decryption.

The first execution of an encrypted code segment bears the penalty of the decryption; therefore, we assume the performance penalty of running encrypted code will be larger. Table 6 shows the duration of the first and single call to FFT in each of the configurations aforementioned.

	Avg	StdDev	Max	Min
Native	4.6	0.54	5	4
Live-decryptc	460	27	480	421
Cache	407	28	456	385
RT-map	476	14	485	451

Table 6 Duration of a single FFT in microseconds

As can be seen, the first call to FFT is time-consuming. In the best case, it is 80 times worse than the reference test, which is five microseconds.

6.3.2 Repeated execution overhead

We measured the time for additional executions of encrypted code beyond the first. In the cached and real-time mapping modes, after the first run, the overhead with additional calls includes the overhead of the context switch to HYP mode but does not include the time required for the decryption, as this work was already completed earlier.

In Table 7 we executed the FFT after the function was decrypted into a temporary buffer (in the cache and RT-map cases). We benchmarked 1,000 consecutive calls to the FFT function.

	Avg	StdDev	Max	Min
Native	888	5	896	884
Live-decrypt	398914	2717	402444	396012
Cache	7323	346	7712	6800
RT-map	928	3	934	926

Table 7 Duration of a 1000 FFT calls in microseconds

From Table 7, we see that the fastest mode is the real-time mapping, and has the smallest deviation. In the live-decrypt mode, the overhead is caused by the constant decryption, when entering the hypervisor, as well as putting back the pad code when exiting the hypervisor back to EL0 and, lastly, flushing the cache. In the cache mode, we can see how the overhead of the repeated decryption impacts the speed. In real-time mapping, we gradually map the process's address space to the hypervisor, so there is no exit from the hypervisor except when the process exits. Because the program is running without any interrupts and exits the hypervisor only a single time, we obtain an execution time close to the Native execution time. We can also see that in real-time mapping execution time is more predictable than the other alternatives.

We, therefore, conclude that it is best to remain in EL2 as much as possible. The cost of constantly decrypting and padding back is significant.

6.4 Predictability

To show the RT-map mode is faster than the Native mode because it runs without interrupts, we performed an additional test. Figure 13 is a CPU-intensive FFT function being executed in a tight loop. We generated a large number of interrupts (approximately 3,000 network interrupts/second, while in idle state is approximately 300 network interrupts/second) and then we executed a simple FFT function one thousand times. The interruptless mode is when we executed FFT through a hypervisor without any decryption. The reference test is when we executed FFT as a standard Linux function.

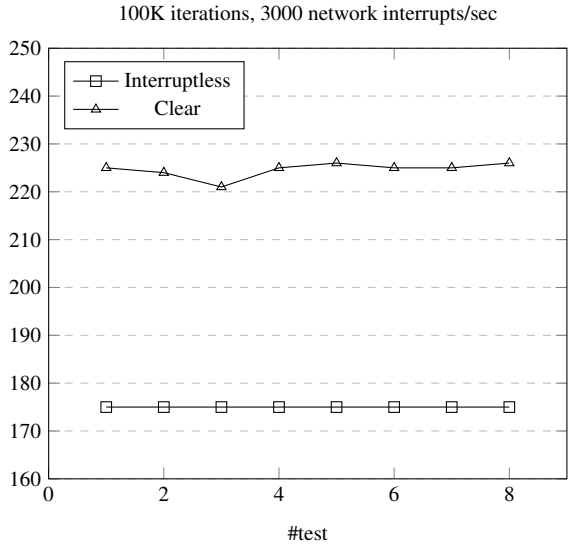


Fig. 13 FFT duration in milliseconds

Figure 13 shows a gap of over 28% between the two runs. We, therefore, conclude that the difference between RT-map to Native in Table 7 is due to the execution conditions.
Next, we want to measure more realistic loads. We measure malloc(3) [41] and free(3) [41], disk IO, file open(2) and close(2), memory access, and all these operations combined.
It is important to note that the proposed system supports the use of any of these operations (malloc(3), open(2), write(2)) in an encrypted context; however, it does not offer to obfuscate them. This system localizes the obfuscation to some functions in a program.
The next tests were performed only in RT-map mode.

6.5 Stack access overhead

In Table 8, we measure a real-life performance of memory access, read, and write when the memory is allocated on the stack. We encrypted a small function so the overhead of decryption and cache eviction would be low.

Iterations	Measure	Encrypted	Clear
1			
	Avg	325	25
	StdDev	43	0.9
	Max	378	26
	Min	280	24
10			
	Avg	328	28
	StdDev	23	1.7
	Max	365	30
	Min	304	26
100			
	Avg	388	58
	StdDev	34	1
	Max	423	60
	Min	351	57
1000			
	Avg	676	377
	StdDev	16	10
	Max	694	390
	Min	661	370

Table 8 Duration of stack access in microseconds

Stack access (Table 8) is an essential test because the stack is being mapped in real-time to the microvisor and the kernel (EL1). In this test, we used a stack of 10 pages. We wanted to evaluate how costly overhead of real-time mapping. As we can see, the first run is 13 times slower than the non-encrypted program with a 14% standard deviation. However, as the number of iterations grows, the overhead mitigates. In 1000 iterations it is 80% with 2% standard deviation. We, therefore, conclude it is best to use a pre-mapped memory, and if possible, pre-map the stack or any other memory that is accessed in the hypervisor.

6.6 A RAM access overhead

In Table 9, we access a heap memory randomly. We did not test the malloc(3) itself but only the memory access: a read and a write.

Iterations	Measure	Encrypted	Clear
1			
	Avg	369	163
	StdDev	27	0.7
	Max	402	164
	Min	342	162
10			
	Avg	1758	1566
	StdDev	28	6
	Max	1791	1578
	Min	1718	1562
100			
	Avg	15551	15481
	StdDev	24	21
	Max	15585	15513
	Min	15529	15459
1000			
	Avg	153461	154338
	StdDev	28	152
	Max	153502	154610
	Min	153439	154247

Table 9 Duration of RAM access in microseconds

In Table 9, we access a large amount of data (1MB), so the first MMU aborts duration is negligible compared to the memory access duration. When running 1000 iterations before exiting the hypervisor, the overhead of exiting the hypervisor is not noticeable.

Evidently, the less we exit the hypervisor, the lower the penalty. In general, it is best to map the heap memory to the hypervisor as early as possible, to reduce MMU aborts to EL2.

6.7 malloc(3)/free(3) overhead

We have benchmarked the standard memory allocator under Linux. In Table 10, we test the cost of malloc(3) and free(3) without accessing memory, i.e., we call malloc(3) and free(3) repeatedly.

Iterations	Measure	Encrypted	Clear
1			
	Avg	224	117
	StdDev	53	7
	Max	279	124
	Min	136	110
10			
	Avg	483	145
	StdDev	21	6
	Max	520	150
	Min	465	135
100			
	Avg	462	161
	StdDev	35	6
	Max	522	169
	Min	434	156
1000			
	Avg	734	409
	StdDev	14	15
	Max	752	430
	Min	713	395

Table 10 Duration of malloc(3)/free(3) access in microseconds

Here (Table 10), the overhead for a single iteration is 200% and is gradually reduced to 80% over 1000 iterations. This code does not perform any page faults as it does not access the allocated memory at all. Like in the FFT and the Stack access tests, we can see that the repeated decrypting, cache and TLB evacuation is approximately 80% for small functions.

In real-time sensitive programs, it is best to avoid malloc(3) and free(3) as much as possible. Because a CPU-bound program is unlikely to perform memory allocations in real time, this overhead can be avoided by using pre-allocation and prematurely mapping the RAM to the hypervisor.

6.8 A File open/close overhead

Table 11 presents measures of I/O performance associated only with opening and closing a file over Linux and the standard ext4 file system. The test opens and closes a single file 1,10...1000 times repeatedly.

Iterations	Measure	Encrypted	Clear
1			
	Avg	170	26
	StdDev	27	1.8
	Max	212	29
	Min	137	25
10			
	Avg	315	84
	StdDev	28	2.1
	Max	345	345
	Min	278	278
100			
	Avg	1070	632
	StdDev	15	7.5
	Max	1094	641
	Min	1057	623
1000			
	Avg	8768	5982
	StdDev	73	33
	Max	8890	6034
	Min	8697	5929

Table 11 Duration of open/close in microseconds

There is an overhead of 30% in favour of running the clear text in 1000 iterations. Like in the previous tests, the overhead decreases as the number of iterations grows, this is because for each open(2) and close(2) the hypervisor exits, and the duration of these system calls is small compared with the decryption of the test function.

In general, we can expect system calls to have some impact on performance due to context switches. We should try to decrease the code that produces system calls as much as possible. For instance, getting the time is extensively used in programs, so it is best to avoid getting the time through a system call but rather by accessing the timer clock register *cntvct_el*0 directly.

6.9 A file write overhead

Table 12 measures IO performances associated with file writes under Linux and, the standard ext4 file system. In Table 12, we measure most of the above operations in addition to file writing operations. The test included memory allocation, file opening and closing, random memory allocation, memory access and memory freeing. The test was performed from a single file, up to 10 files.

#Files	Measure	Encrypted	Clear
1			
	Avg	3513	4497
	StdDev	417	158
	Max	4166	4769
	Min	3121	4395
2			
	Avg	9135	8494
	StdDev	418	200
	Max	9880	8849
	Min	8910	9381
3			
	Avg	11758	11876
	StdDev	1492	470
	Max	13577	12715
	Min	9885	11595
4			
	Avg	15724	15208
	StdDev	158	660
	Max	15965	16387
	Min	15552	14862
5			
	Avg	19130	18235
	StdDev	138	167
	Max	19280	18376
	Min	18983	17945
6			
	Avg	22678	21674
	StdDev	209	43
	Max	23006	21707
	Min	22488	21599
7			
	Avg	25913	25875
	StdDev	139	1378
	Max	26046	27419
	Min	25684	24766
8			
	Avg	29305	29220
	StdDev	146	1531
	Max	29475	30906
	Min	29142	28000
9			
	Avg	33273	32061
	StdDev	1253	1347
	Max	35453	34432
	Min	32410	31173
10			
	Avg	33156	34890
	StdDev	2402	214
	Max	38084	35099
	Min	30005	34563

Table 12 Duration of IO write in microseconds

It is noticeable that the more IO is processed, the less the difference between the two executions. In Table 12, the effect of running our security hypervisor is unnoticeable. Encryption works with a negligible overhead in most cases. The decryption and cache invalidation penalties are negligible compared to the long duration of the IOs.

7 Future work

We intend to examine [48] on the ARM platform. This method offers performance benefit on Intel architecture, and we intend to examine it on ARM architecture as well.

We expect further work to be undertaken in the ARM microvisor area.
We intend to utilize the microvisor in other ways. The hyplet presented in this paper is rapidly evolving in new directions. To name a few, we present the hyplet as a means to run userspace interrupts without overhead in Linux, and as an extremely fast remote procedure call (RPC). C-FLAT [44] - a control attestation system for embedded systems, was developed for TrustZone in devices with a minimal operating system. We will present an innovative technique to run C-FLAT in Linux with our new RPC. Kiperberg et al [45] presents a hypervisor-assisted atomic memory acquisition for the x86 architecture, we intend to present a port for hypervisor memory acquisition tool in ARM through the use of a microvisor.

The offline scheduler [40] is a technique to execute programs in an unplugged processor in Linux. We intend to demonstrate an evolution of the offline scheduler in the form of the offline microvisor.

8 Summary

This paper is a proof of concept that reverse engineering protection in ARM is applicable for CPU intensive workloads with minimal overhead. We achieved that by minimizing the number of context switches between the hypervisor and EL0. We can also assume that the encrypted sections are significant, and as such, the padding and the decrypting takes longer as the function size increases. For this, we recommend to minimize system calls and pre-map any memory that is accessed in the HYP context. For I/O intensive programs, we showed that the encryption penalty is relatively small compared to the I/O penalty, so our technology is most suitable for programs with high I/O rate.
Our solution proved stable during our internal testing. However, we also note that our ARM-based technology has not passed the same level of stability testing and penetration testing that the Intel solution has.

9 Compliance with Ethical Standards

Raz Ben Yehuda and Nezer Jacob Zaidenberg both declare that they own stock in TrulyProtect.
Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Victor Costan and Srinivas Devadas, Intel sgx explained, IACR Cryptology ePrint Archive, 2016:86, 2016.

2. Balaji Balakrishnan, Matthew Hosburgh, and Patrick Neise, Securing the windows 10 GIAC enterprise endpoint.
3. Amir Averbuch Michael Kiperberg and Nezer Jacob Zaidenberg, Truly-protect: An efficient VM-based software protection. IEEE Systems Journal, 7(3):455–466, 2013
4. Niels Penneman Danielius Kudinskas Alasdair Rawsthorne Bjorn De Sutter and Koen De Bosschere, Formal virtualization requirements for the arm architecture, Journal of Systems Architecture, 144 - 154, 2013
5. Shaked Flur Kathryn E Gray Christopher Pulte Susmit Sarkar Ali Sezgin, Luc Maranget Will Deacon and Peter Sewell, Modelling the ARMv8 architecture, operationally: concurrency and ISA, In ACM SIGPLAN Notices, volume 51, pages 608–621. , 2016
6. HM Cantero S Peter and Segher Bushing, Console hacking 2010–ps3 epic fail, Chaos Communication Congress (December 2010), 2010.
7. Kevin Coogan Gen Lu and Saumya Debray, Deobfuscation of virtualization-obfuscated software: a semantics-based approach, In Proceedings of the 18th ACM conference on Computer and communications security, pages 275–284. ACM, 2011.
8. Rutkowska Joanna, Introducing blue pill, The official blog of the invisible things, volume 22, pages 23, 2006
9. Eldar Avigdor and Herbert Howard C and Goel Purushottam and Blumenthal Uri and Hines David and Smith Carey, Provisioning active management technology (AMT) in computer systems, Google Patents, US Patent 8 438 618, 2013
10. SMM loader and execution mechanism for component software for multiple architectures, Zimmer, Vincent J, Google Patents, 2005, US Patent 6848046
11. Winter Johannes, Proceedings of the 3rd ACM workshop on Scalable trusted computing, Trusted computing building blocks for embedded Linux-based ARM trustzone platforms, pages= 21–30, 2008, ACM
12. Winter Johannes, Trusted computing building blocks for embedded Linux-based ARM trustzone platforms, Proceedings of the 3rd ACM workshop on Scalable trusted computing, pages=21–30, 2008, ACM
13. Heiser Gernot and Leslie, Ben, The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors, Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems, 19–24, 2010
14. Bernard Ngabonziza Daniel Martin Anna Bailey Haehyun Cho and Sarah Martin, Trustzone explained: Architectural features and use cases, Collaboration and Internet Computing (CIC), 2016 IEEE 2nd, International Conference on, pages 445–451. IEEE, 2016.
15. Amit Resh Michael Kiperberg Roee Leon and Nezer Zaidenberg, System for executing encrypted native programs, International Journal of Digital Content Technology and its Applications, 11, 2017.
16. Amit Resh Michael Kiperberg Roee Leon and Nezer J Zaidenberg, Preventing execution of unauthorized native-code software. International Journal of Digital Content Technology and its Applications, 11, 2017.
17. William Rosenblatt Stephen Mooney and William Trippe, Digital rights management: business and technology, John Wiley & Sons, Inc., 2001.
18. Gernot Heiser and Ben Leslie, The okl4 microvisor: Convergence point of microkernels and hypervisors, Proceedings of the first ACM Asia-Pacific workshop on Workshop on systems, pages 19–24. ACM, 2010
19. Thom, Stefan, Jeremiah Cox, David Linsley, Magnus Nystrom, Himanshu Raj, David Robinson, Stefan Saroiu, Rob Spiger, and Alastair Wolman. "Firmware-based trusted platform module for arm processor architectures and trustzone security extensions." U.S. Patent 8,375,221, issued February 12, 2013.
20. Nikolay Elenkov, Android security internals: An in-depth guide to Android's security architecture, No Starch Press, 2014.
21. Dan Rosenberg, QSEE trustzone kernel integer overflow vulnerability, Black Hat conference, page 26, 2014.
22. Prerna Mahajan and Abhishek Sachdeva, A study of encryption algorithms AES, DES and RSA for security, Global Journal of Computer Science and Technology, 2013
23. Amir Moradi, Mohammad T Manzuri Shalmani, and Mahmoud Salmasizadeh, A generalized method of differential fault attack against AES cryptosystem, International Workshop on Cryptographic Hardware and Embedded Systems, pages 91–100. Springer, 2006
24. Darrel Hankerson, Alfred J Menezes, and Scott Vanstone, Guide to elliptic curve cryptography, Springer Science & Business Media, 2006.
25. Asaf David and Nezer Zaidenberg, Maintaining streaming video DRM, In Proceedings of The International Conference on Cloud Security Management ICCSM-2014, page 36, 2014.
26. Marc Eisenstadt and Mike Brayshaw, The transparent prolog machine (TPM): an execution model and graphical debugger for logic programming, The Journal of Logic Programming, 5(4):277–342, 1988.
27. Rick Kennell and Leah H Jamieson, Establishing the genuinity of remote computer systems, In USENIX Security Symposium, pages 295–308, 2003
28. Michael Kiperberg and Nezer Zaidenberg, Efficient remote authentication, In Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013, page 144. Academic Conferences Limited, 2013.

29. Kari Kostiainen N Asokan and Jan-Erik Ekberg, Practical property-based attestation on mobile devices, In International Conference on Trust and Trustworthy Computing, pages 78–92. Springer, 2011.
30. Karsten Sohr Tanveer Mustafa and Adrian Nowak, Software security aspects of java-based mobile phones, In Proceedings of the 2011 ACM Symposium on Applied Computing, pages 1494–1501. ACM, 2011.
31. Haryadi S Gunawi, Mingzhe Hao Tanakorn Leesatapornwongsa Tiratat Patana-anake Thanh Do Jeffry Adityatama Kurnia J Eliazar Agung Laksono Jeffrey F Lukman Vincentius Martin, et al, What bugs live in the cloud? a study of 3000+ issues in cloud systems. In Proceedings of the ACM Symposium on Cloud Computing, pages 1–14. ACM, 2014.
32. Amit Vasudevan Jonathan M McCune and James Newsome, Trustworthy execution on mobile devices, Springer, 2014
33. Jinmok Kim Donguk Kim Jinbum Park Jihoon Kim and Hyoungshick Kim, An efficient kernel introspection system using a secure timer on trustzone. Journal of the Korea Institute of Information Security and Cryptology, 25(4):863–872, 2015.
34. Min Zhu Bibo Tu Wei Wei and Dan Meng HA-VMSI, A lightweight virtual machine isolation approach with commodity hardware for ARM, In Proceedings of the 13th ACM SIGPLAN/SIGOPS International, Conference on Virtual Execution Environments, pages 242–256. ACM, 2017
35. Nelson Elhage, Virtualization under attack: Breaking out of KVM, DEF CON, 19, 2011.
36. Peter Szor. The art of computer virus research and defense, Pearson Education, 2005.
37. Christoffer Dall and Jason Nieh, KVM/ARM: the design and implementation of the Linux ARM hypervisor, ACM SIGARCH Computer Architecture News, 42(1):333–348, 2014.
38. Marcan Sven and Comex, Console hacking 2013–u fail it, In 30th Chaos Communication Congress (December 2013), 2013.
39. Roberto Mijat and Andy Nightingale, Virtualization is coming to a platform near you. ARM white paper, 20, 2011.
40. Ben-Yehuda and Wiseman(2013) The offline scheduler for embedded vehicular systems, International Journal of Vehicle Information and Communication Systems, Volume 3 pages 44–57
41. Maurice J Bach et al, The design of the UNIX operating system, volume 1, Prentice-Hall Englewood Cliffs, NJ, 1986.
42. Anatoli Kalysch, Johannes Götzfried, and Tilo Müller, Vmattack: Deobfuscating virtualization-based packed binaries. In Proceedings of the 12th International Conference on Availability, Reliability and Security, page 2. ACM, 2017
43. Jan-Erik Ekberg Kari Kostiainen and N Asokan, The untapped potential of trusted execution environments on mobile devices, IEEE Security & Privacy, 12(4):29–37, 2014.
44. Abera, Asokan, Davi, Ekberg, Nyman, Paverd, Sadeghi, and Tsudik. C-flat: control-flow attestation for embedded systems software, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 743-754, 2016
45. Kiperberg, Leon, Resh, Algawi, and Zaidenberg, Hypervisor-assisted atomic memory acquisition in modern systems,
46. Sebastian Banescu Ciprian Lucaci Benjamin Krämer, and Alexander Pretschner, Vot4cs: A virtualization obfuscation tool for c. In Proceedings of the 2016 ACM Workshop on Software PROtection, pages 39–49. ACM, 2016.
47. Boaz Barak Oded Goldreich Rusell Impagliazzo Steven Rudich Amit Sahai Salil Vadhan and Ke Yang, On the (im) possibility of obfuscating programs, In Annual International Cryptology Conference, pages 1–18. Springer, 2001.
48. Kiperberg, Michael, Roee Leon, Amit Resh, Asaf Algawi, and Nezer J. Zaidenberg. "Hypervisor-based Protection of Code." IEEE Transactions on Information Forensics and Security (2019).

PVI

THE HYPLET-JOINING A PROGRAM AND A NANOVISOR FOR REAL-TIME AND PERFORMANCE

by

Raz Ben Yehuda, Nezer Zaidenberg 2020

International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). IEEE

The hyplet - Joining a Program and a Nanovisor for real-time and Performance

Raz Ben Yehuda
University of Jyväskylä
Jyväskylä, Finland
raziebe@gmail.com

Nezer Jacob Zaidenberg
College of Management Academic Studies
Israel
scipio@scipio.org

***Abstract*—This paper presents the concept of sharing a hypervisor address space with a standard Linux program. In this work, we add hypervisor awareness to the Linux kernel and execute code in the HYP exception level through using the hyplet. The hyplet is an innovative way to code interrupt service routines and remote procedure calls under ARM. The hyplet provides high performance and run-time predictability. We demonstrate the hyplet implementation using the C programming language on an ARM8v-a platform and under the Linux kernel. We then provide performance measurements, use cases, and security scenarios.**

***Index Terms*—Hypervisor, real time, Linux, Virtualization, Security**

I. Introduction

There are various techniques to achieve real-time computing. One is to use a single operating system that provides real-time computing, such as standalone VxWorks or RT PREEMPT which is a Linux kernel extension. Another technique is the microvisor that co-exist with the general purpose operating system. A microvisor is an operating system that employs some characteristics of a hypervisor and some characteristics of a microkernel. A typical architecture of a microvisor. OKL4 [8] is an example for operating system that uses a microvisor. The hyplet (Figure 1) is a software, code and data, shared between a process and a nanovisor. It is a hybrid of a normal user program and a nanovisor that offers real-time processing and performance. A hyplet program (1) maps parts of code and data to the nanovisor, then it associates (2) the hyplet handler with an event, IRQ or RPC (or both). At this point (3), RPC that traps to the nanovisor, or IRQ (4) that upcalls the nanovisor would trigger an (5) hyperupcall to the hyplet.

We introduce the hyplet for the purpose of interrupt handling in user-space and for the purpose of efficient interprocess communication. The fast RPC and the user-space interrupts are done in a standard Debian, and with little modifications to the user program. This paper aims to provide Real-Time responsiveness to interrupts and fast RPC in cases where it is not cost-effective to run a standalone RTOS or Linux RT PREEMPT.

We present the hyplet ISR (Interrupt Service Routine) as a technique to reduce hardware to user-space latency. The hyplet is not aimed to replace a kernel space drivers in user-space, but to propagate the interrupt to a process. Thus, the hyplet can affect a driver's behavior. For example, should the driver process the packet or not. We will use the term hypISR to distinguish a normal ISR from an ISR in hyplet mode.

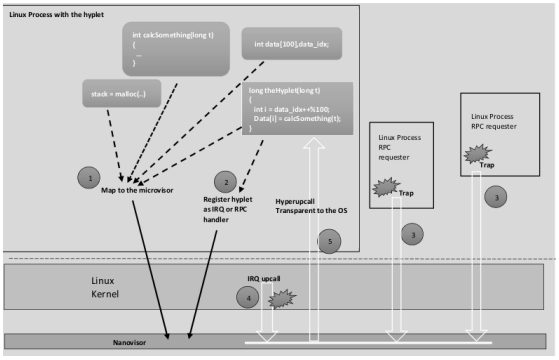


Fig. 1. The hyplet nanovisor

In addition to hypISR, we present hypRPC. HypRPC is a reduced RPC mechanism which has a latency of a sub-microsecond on average, and 4 microseconds worst case. Our RPC is a type of a hypervisor trap where the user process sends a procedure id to the hypervisor to be executed with high privilege without interrupts in another process address space. We use the term hypRPC for our RPC as a mixture between hyperuprcall and RPC.

One motivation for hypRPC is safety. In many cases, Real-Time programmers tend to encapsulate most of the software functionality in a single address space, so that the various threads would easily access shared data. However, this comes with the cost of a single thread crashing the entire process in case of an error in this thread. Through the use of the hypRPC we can separate a single multi-threaded process to multiple processes. Other RPC solutions, as we show later in the paper, are slower.

The hyplet is based on the concept of a delicate address space separation within a process. Instead of running multiple operating systems kernels, the hyplet divides the Linux process into **two execution modes: HYP and Normal**. Part of the process (HYP mode) would execute in an isolated, non-interrupted privileged safe execution environment. The other part of the process would execute in a regular user mode (Normal mode). To summarize, the hyplet is meant to reduce the latency of hardware interrupt to a user-space program,

and program to program local communication in user-space programs to sub microsecond order of magnitude.
In the taxonomy of virtualization, hyplets are classified as bare metal type 2 hypervisors. A type 2 hypervisor is a hypervisor that is loaded by the host operating system. A type 1 hypervisor is a hypervisor which is loaded by the boot loader, prior to the general operating system (GPOS). The hyplet does not require to be a virtual machine; thus it may execute in hardware that does not have support for interrupts virtualization. It is meant to be simple to use and adapt to the existing code. It does not require any modifications to the boot loader, only minor changes to the Linux kernel.

II. Background

ARM has a unique approach to security and privilege levels that is crucial to the implementation of the hyplet. In ARMv7, ARM introduced the concept of secured and non-secured worlds through the implementation of TrustZone, and starting from ARMv7a. ARM presents four exception (permission) levels as follows.
Exception Level 0 (EL0) Refers to the user-space code. Exception Level 0 is analogous to ”ring 3” on the x86 platform.
Exception Level 1 (EL1) Refers to operating system code. Exception Level 1 is analogous to ”ring 0” on the x86 platform.
Exception Level 2 (EL2) Refers to HYP mode. Exception Level 2 is analogous to ”ring -1” or ”real mode” on the x86 platform.
Exception Level 3 (EL3) Refers to TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no direct analogous modes, but related concepts in x86 are Intel’s ME or SMM.

Each exception level provides its own set of special purpose registers and can access these registers of the lower levels, but not higher levels. The general purpose registers are shared.
Microvisors, Microkernels, virtualization and para virtualizion are all possible in this architecture. Microvisors are operating systems that execute in EL2, Microkernels virtual memory management (and some other parts) are kept mostly in EL1, while the user services are kept in EL0. Virtualization is kept in EL2 and para virtualization is kept both in EL1 and EL2.

III. The hyplet

The hyplet is a native C function that runs in a nanovisor and a Linux process. It does not require any special compilation or pre-processing. But before diving into the technical details, we describe our motivation through use cases.
Trusted Interrupts
The hyplet can be used to mask the handling of an interrupt so that it will not be visible by the OS driver. Interrupts handled by the hyplet can be verified by TPM or TrustZone, and pose an extra layer of protection to reverse or modify, unlike OS-based interrupt handler.
To modify a normal OS interrupt it is sufficient to elevate privileges to the OS level. To modify a hyplet one must first elevate permissions to the OS level, and then attack and subvert the hypervisor itself.

The hyplet a malware detector
We showed that the hyplet RPC is the fastest in Linux. For this reason, we used this technology for C-FLAT [1]. C-FLAT is a run time remote attestation technique that detects malware-infected devices. It does so by recording a program runtime behavior and monitoring the execution path of an application running on an embedded device. [1] presents C-FLAT through the use of the TrustZone. We implemented C-FLAT through the use of hypRPC. We replaced the various branch opcodes with the trap opcode. This effort is completed, and due to the low overhead of the hypRPC, we can present this technology in Linux and not in bare metal as in [1].

Protection against reverse engineering
On x86 platforms, TrulyProtect provides anti-reverse engineering , end-point security, video decoding, forensics etc. TrulyProtect relies on Dynamic Root of Trust Measurement (DRTM) attestation to create a trusted environment in the hypervisor to receive encryption keys [12]. We have used the hyplet to implement a TrulyProtect-like system on the ARM platform. We have encrypted parts of the software and used the hyplet in order to switch context and elevate privileges. Our systems then decode the code in the hypervisor context (a hyplet), so that the code or decryption keys will not be available to the OS. Our system for protection against reverse engineering has a cost affiliated with first execution and decryption of the code, but very low per iteration overhead as demonstrated in the table below.

Iterations	Encrypted	Clear
1	1185	1127
10	2737	2597
100	18022	18018
1000	173925	171251
10000	1758997	1670811

TABLE I
Duration of stack access in ticks

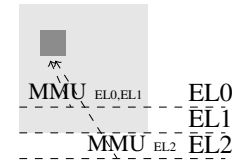
A. The hyplet explained

ARM8v-a specifications offer to distinct between user-space addresses and kernel space addresses by the MSB (most significant bits). The user-space addresses of Normal World and the hypervisor use the same format of addresses.
These unique characteristics are what make the hyplet possible. The nanovisor can execute user-space position-independent code without preparations. Consider the code snippet at Figure 2. The ARM hypervisor can access this code’s relative addresses (adrp), stack (sp_el0) etcetera without pre-processing. From the nanovisor perspective, Figure 2 is a native code. Here, for example, address 0x400000 is used both by the nanovisor and the user.

```
400610: foo:
400614: stp x16, x30, [sp,#-16]!
400618: adrp x16, 0x41161c
40061c: ldr x0, [sp,#8]
400620: add x16, x16, 0xba8
400624: br x17
400628: ret
```

Fig. 2. A simple hyplet

So, if we map part of a Linux process code and data to a nanovisor it can be executed by it.
When interrupt latency improvement is required, the code is frequently migrated to the kernel, or injected as the eBPF framework suggests [5]. However, kernel programming requires a high level of programming skills, and eBPF is restrictive. A different approach would be to trigger a user-space event from the interrupt, but this would require an additional context switch. A context switch in some cases is time-consuming. We show later that a context switch is over 10 μs in our evaluation hardware. To make sure that the program code and data are always accessible and resident, it is essential to disable evacuation of the program's translation table and cache from the processor. Therefore, we chose to constantly accommodate (cache) the code and data in the hypervisor translation registers in EL2 cache and TLB. To map the user-space program, we modified the Linux ARM-KVM, [6] mappings infrastructure to map a user-space code with kernel space data.



Two exception Levels access the same physical frame with the same virtual address of some process. However, the page tables of the two exception levels are not identical.

Fig. 3. Asymmetric dual view

Figure 3 demonstrates how identical addresses may be mapped to the same virtual addresses in two separate exception levels. The dark shared section is part of EL2 and therefore accessible from EL2. However, when executing in EL2, EL1 data is not accessible without previous mapping to EL2. Figure 3 presents the leverage of a Linux process from two exception levels to three.
The natural way of memory mapping is that EL1 is responsible for EL1/EL0 memory tables and EL2 is responsible for its memory table, in the sense that each privileged exception level accesses its memory tables. However, this would have put the nanovisor at risk, as it might overwrite or otherwise garble its page tables. As noted earlier, on ARM8v-a hypervisor has a single memory address space. (unlike TrustZone that has two, for kernel and user). The ARM architecture does not coerce an exception level to control its memory tables. This makes it possible to map EL2 page table in EL1. Therefore, only EL1 can manipulate the nanovisor page tables. We refer to this hyplet architecture as a Non-VHE hyplet. Also, to further reduce the risk, we offer to run the hyplet in injection mode. Injection mode means that once the hyplet is mapped to EL2, the encapsulating process is removed from the operating system kernel, but its hyplet's pages are not released from the nanovisor, and the kernel may not re-acquire them. It is similar to any dynamic kernel module insertion.
In processors that support **VHE** (Virtual Host Extension), EL2 has an additional translation table, that would map the kernel address space. In a VHE hyplet, it is possible to execute the hyplet in the user-space of EL2 without endangering the hypervisor. A hyplet of a Linux process in $EL0_{EL1}$ (EL0 is EL1 user-space) is mapped to $EL0_{EL2}$ (EL2 user-space). Also, the hyplet can't access EL2 page tables because the table is accessible only in the kernel mode of EL2. VHE resembles TrustZone as it has two distinct address spaces, user and kernel. Operating systems such as QSEE (Qualcomm Secure Execution Environment) and OP-TEE [18] are accessed through an upcall and execute the user-space in TrustZone. Unfortunately, at the time of writing, only modern ARM boards offer VHE extension (ARMv8.2-a) and therefore this paper demonstrates benchmarks on older boards.

B. The hyplet security & Privilege escalation in RTOS

As noted, VHE hardware is not available at the time of this writing, and as such we are forced to use software measures to protect the hypervisor. On older ARM boards it can be argued that a security bug at hypervisor privilege levels may cause greater damages compared to a bug at the user process or kernel levels thus poising system risk.
The hyplet also escalates privilege levels, from exception level 0 (user mode) or 1 (OS mode) to exception level 2 (hypervisor mode). Since the hyplet executes in EL2, it has access to EL2 and EL1 special registers. For example, the hyplet has access to the level 1 exception vector. Therefore, it can be argued that the hyplet comes with security cost on processors that do not include ARM VHE.
The hyplet uses multiple exception levels and escalates privilege levels. So, it can be argued that using hyplets may damage application security. Against this claim, we have the following arguments.
We claim that this is risk is superficial and an acceptable risk, for processors without VHE support. Most embedded systems and mobile phones do not include a hypervisor and do not run multiple operating system.
In the case where no hypervisor is installed, code in EL1 (OS) has complete control of the machine. It does not have lesser access code running in EL2 since no EL2 hypervisor is present. Likewise code running in EL2 can affect all operating systems running under the hypervisor. Code running in EL1 can only affect the current operating system. When only one OS is running the two are identical.

Therefore, from the machine standpoint, code running in EL1 when EL2 is not present has similar access privileges to code running in EL2 with only one OS running, as in the hyplet use case.
The hyplet changes the system from a system that includes only EL0 and EL1 to a system that includes EL0, EL1, and EL2. The hyplet system moves a code that was running on EL1 without a hypervisor to EL2 with only one OS. Many real-time implementations move user code from EL0 to EL1. The hyplet moves it to EL2, however, this gains no extra permissions, running rogue code in EL1 with no EL2 is just as dangerous as moving code to EL2 within the hyplet system. Additionally, it is expected that the hyplet would be a signed code; otherwise, the hypervisor would not execute it.
The hypervisor can maintain a key to verify the signature and ensure that lower privilege level code cannot access the key.
Furthermore, Real-time systems may eliminate even user and kernel mode separation for minor performance gains. We argue that escalating privileges for real performance and Real-time capabilities is an acceptable on older hardware without VHE where hyplets might consist of a security risk. On current ARM architecture with VHE support the hyplet do not add extra risk.

C. Static analysis to eliminate security concerns

Most memory (including EL1 and EL2 MMUs and the hypervisor page tables) is not mapped to the hypervisor. The non-sensitive part of the calling process memory is mapped to EL2. The hyplet does not map (and thus has no access to) kernel-space code or data. Thus, the hyplet does not pose a threat of unintentional corrupting kernel's data or any other user process unless additional memory is mapped or EL1 registers are accessed.
Thus, it is sufficient to detect and prevent access to EL1 and EL2 registers to prevent rogue code affecting the OS memory from the hypervisor. We coded a static analyzer that prevents access to EL1 and EL2 registers and filters any special commands.
We borrowed this idea from eBPF. The code analyzer scans the hyplet opcodes and checks that are no references to any black-listed registers or special commands. Except for the clock register and general-purpose registers, any other registers are not allowed. The hyplet framework prevents new mappings after the hyplet was scanned to prevent malicious code insertions. Another threat is the possibility of the insertion of a data pointer as its execution code (In the case of SIGBUS of SEGFAULT, the hyplet would abort, and the process terminates). To prevent this, we check that the hyplet's function pointer, when set, is in the executable section of the program.
Furthermore, the ARM architecture features the TrustZone mode that can monitor EL1 and EL2. The TrustZone may be configured to trap illegal access attempts to special registers and prevent any malicious tampering of these registers.

D. The hyplet - User-Space Interrupt

In Linux and other operating systems, when an interrupt reaches the processor, it triggers a path of code that serves the interrupt. Furthermore, in some cases, the interrupt ends up waking a pending process.
The hyplet is designed to reduce the time from the interrupt event to the program. To achieve this, as the interrupt reaches the processor, instead of executing the user program code in EL0 after the ISR (and sometimes after the bottom half), a special procedure of the program (Figure 4) is executed in HYP mode at the start of the kernel's ISR.

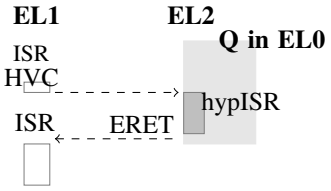


Fig. 4. HypISR flow

The hyplet does not change the processor state when it is in interrupt; thus, once the hyplet is completed, the kernel interrupt can be processed.
The hyplet does not require any new threads and because the hyplet is an ISR, it can be triggered in high frequencies. As a result, we can have high-frequency user-space timers in small embedded devices.
Some may argue that the hyplet should have been implemented as a virtual interrupt. However, many ARMv8-a platforms do not support VGIC (virtual Interrupt Controller). Raspberry PI3, for example, does not fully support VGIC (as a consequence, ARM-KVM does not run on a Raspberry PI3). Interrupts are then routed to the hypervisor by upcalls from the kernel main-interrupt routine, and the nanovisor communicates with the guest through a hyperupcall [2]. Nested hyplet interrupts are not possible.

E. Hypervisor based RPC

The remote procedure call is a type of interprocess communication (IPC) in which parameters are transferred in the form of function arguments. The response is returned as the function return value. The RPC mechanism handles the parsing and handling of parameters and the returned values. In principle, RPC can be used locally as a form of IPC and remotely over TCP/IP network protocols. In this paper, we only consider the local case.

IPC in real-time systems is considered a latency challenge. Thus system developers refrain from using IPC in many cases. The solution many programmers use is to put most of the logic in a single process. This technique decreases the complexity but increases the program size and risks.
In multicore computers, one reason for the latency penalty is because the receiver may not be running when the message is sent. Therefore, the processor needs to switch contexts.

HypRPCs are intended to reduce this latency to the sub-microsecond on average by eliminating the context switch (in some way the hyplet is viewed as a temporary address-space extension to the sending program).
If the receiving program exits, then the API immediately returns an error. If the function needs to be replaced in real-time, there is no need to notify the sending program; instead, the function in the hypervisor only needs to be replaced.

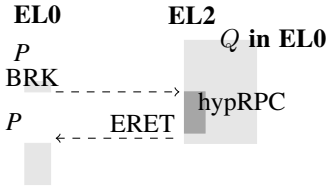


Fig. 5. HypRPC flow

Figure 5 demonstrates the hypRPC flow. Program P is a requesting program, and Q is a serving program. As program P loads, it registers itself as a hypRPC requesting program. A hypRPC program, unlike hypISR, is a program that when it executes the BRK instruction, it traps into HYP mode. The reason for that is that user-space programs are not permitted to perform the upcall instruction. When P calls BRK, the first argument is the RPC id, i.e; $x_0 = rpc_{id}$. As the processor executes BRK, it shifts to HYP mode. Then the hypervisor checks the correctness of the caller P and the availability of Process Q, and if all is ok, it executes in EL2 the function with this id.
The semantics of hypRPC is different from the common RPC. In a common RPC, the receiver is required to assign a thread to wait for the caller, and if a single thread is used, a serialized access provides protection. In Q, accessing a resource shared from the nanovisor and Normal world must be protected, even if Q has no threads. The protection is achieved by disabling context-switch and disabling access from another processor, using synchronization primitives we provide.

The hyplet has some additional benefits:
Safety
The hypISR provides a safe execution environment for the interrupt. In Linux, if there is a violation while the processor is in kernel mode, the operating system may stop. In the hyplet case, if there is a fault in the hypISR, the nanovisor would trigger a violation (for instance, a SEGFAULT). The nanovisor would send, through the kernel, a signal to the process containing the hyplet. This signal is possible because the fault entry of the nanovisor handles the error as if it is a user-space error. For example, if a divide-by-zero failure happens, the operating system does not crash, but the hyplet capable-program exits with a SIGFPE.
Some may claim that endless loops may hog the processor. For this, we argue that OP-TEE shares the same vulnerability because in OP-TEE the tee-supplicant blocks on a session until the TA (trusted application) finishes. However, we still offer to handle endless loops by replacing the hyplet code by the NOP (no operation) opcode. The hyplet will exit back to the nanovisor, the nanovisor will put back the original code and send a SIGKILL signal to the process.
We offer to handle code injection by caching the original code and overwriting it once an injection is detected. It is possible to checksum the hyplet code and see if it is changed, and if so re-write the original code [15].
Another facet of hypISR is sensitive data protection. We can use the hyplet to securely access data, and I/O data may be hidden from EL1 and accessible only in EL2.
Scope of Code Change
The hyplet patch does not interfere with the hardcore of the kernel code, and the hyplet patch does not require any modification to any hardware driver. The modifications are in the generic ISR routine. As a result, it is easy to apply this technology as it does not change the operating system heuristics. Microvisors, such as seL4 and Xvisor, are not easily applied to arbitrary hardware because they require a modified boot loader, making it impossible to apply the microvisors in some cases, for example, when the boot-loader code is closed (mobile phones). Jailhouse [3] and KVM do not even run on many devices because virtualization does not suffice (the GIC virtualization is incompatible) in some cases, and raspberry PI3 is an example of such virtualization hardware. In Android OS, it is undesirable to apply RT PREEMPT because it changes the entire operating system behavior. Our nanovisor does provide any service other than bridging to the hyplet. A user chooses to add services to the microvisor as part of the hyplet-utils library.

IV. Evaluation

We demonstrate that the hyplet is suitable for hard real-time systems. We provide synthetic microbenchmarks, and compare our solution to Normal Debian Linux, RT PREEMPT Linux(kernel version 4.4.92), seL4 microkernel (version 10.0.0.0) [10], and Xvisor (v0.2.11), all on a Raspberry PI3. PI3 main specifications are a 19.2 MHz, 4 ARM Cortex A53 1.2GHz cores and 1GB RAM 900 MHz.
We selected Xvisor because Xvisor is a thin microvisor. We chose RT PREEMPT because it is considered a free open source non-commercial RTOS Linux OS, and we chose seL4 because it is a hard real-time mathematically proven microkernel. The time units we use are due to the clock granularity, which is $\frac{10^9}{19.2*10^6} = 52$ nanoseconds per tick. We start by evaluating PI3 interrupt latency.

A. Interrupt Latency

To understand the possible time deviations in the Timer's test(in the next section), we start first by evaluating the latency of an interrupt in PI3.
We measured the delay from the attached hardware to the start of the hyplet. For this purpose, we connected an Invensense

mpu6050 to the PI, and configured this IMU to work in i2c protocol. In i2c, for every 8 bits of data, there is an acknowledgment signal, that generates an interrupt to the PI. We wanted to measure the time interval between the moment of the i2c ACK, to the moment the processor runs the main interrupt routine. So, we connected a logic analyzer probe to the SDA of the IMU and programmed one of the PI's GPIO to trigger a signal in the kernel's main interrupt routine. This way we could take the time of the IMU ACK signal, and the kernel ISR time. We generated the i2c signals in random times. The results were an average of 3.9 μs, a maximum of 9μs, and the minimum was 1.7μs.
First, we should not expect interrupts to be processed in deterministic times. A deviation of nearly 5 μs from the average (3.9 μs) is a lot. This can happen for various reasons, such as interrupts congestion or TLB latency and so on.
Also, this benchmark means that if we connect a device that ticks in a high frequency, such as 100Khz, two consequent interrupts may appear in 1.9μ delta. So, while the kernel can handle these interrupts in-accuracy, the user-space will miss the second interrupt.

B. Timer

We continue the evaluation and construct a hyplet timer. Table II presents the measured delay latencies of the timer programs in various operating systems. We conducted a delay of 1 ms for 5 minutes. In RT PREEMPT and Normal Raspbian Linux we used cyclictest, a real-time test suit for Linux. Cyclictest is a test software that measures timer latency in Linux. Cyclictest implements a sleeper thread, takes a time sample, goes to sleep, and when woken it records the time differences and goes back to sleep again. Cyclictest binds the waiting thread to a single processor.
Since Cyclictest is not available in seL4 and Xvisor, we wrote a timer microbenchmark that sleeps for 1 ms and records the time differences. In Xvisor, to make the test equal to the hyplet as much as possible, in terms of which privilege level the code was executed, our simple timer ran in HYP mode (not in the VM/guest OS).

ranges in μs	RT_ PRPT	Hyplet	Nrmal	Xvsr	seL4
0	0	99.9477	0	0	0
1	0	0.0523	0	0	0
2-5	0	0.0020	0	0	100
6-10	0	0	47.7	99.9	0
11-15	69	0	49.7	0	0
16-20	28	0	1.6	0	0
21-25	2	0	0.25	0	0
26-30	0.085	0	0.26	0	0
31-35	0.01	0	0.0874	0	0
36-40	0.05	0	0.034	0	0
41-45	0.001	0	0.034	0	0
46-50	0.0003	0	0.05	0	0
51-55	0	0	0.0321	0	0
56-100	0	0	0.18	0	0
101+	0	0	0.0014	.1	0

TABLE II
: LATENCIES DISTRIBUTION IN PERCENTAGE

Table II presents the delay deviations of each OS. The tests were conducted while the operating systems were idle. This is because it is not easy to load the operating systems and measure the load in all the operating systems we tested, and in some cases, the operating systems were not stable enough to sustain a load. For the analysis, we assume a deviation of approximately 5% soft real-time and below hard real-time.
In the hyplet case, 99.96% of the samples are below 1μs latency, and 100% are below 5μs. The deviation is probably because of the interrupt latency we showed earlier. The maximum latency of 9 μs is probably not reflected here, because, in this test, the interrupt source is the local timer. The deviation is below $\frac{5}{1000} = \frac{1}{2}\%$.
In the RT PREEMPT case, the upper boundary was 47μs, and 14μs on average. RT PREEMPT's deviation in PI3 is nearly 5% = $\frac{50}{1000}$. We consider RT PREEMPT on PI3 soft real-time.
In Normal Linux, the maximum value was 144μs, and the distribution of the values was higher. So the deviation is 14 %, which, as expected, shows that Normal Linux is a non-real-time OS.
Xvisor presents an impressive benchmark where 99.9% of samples jitter is less than 8 μs, the rest, unfortunately, were nearly 500 μs. Xvisor is not RTOS.
SeL4 is an RTOS. SeL4 produces a remarkable latency of less than 5 μs, approximately $\frac{1}{2}$ % maximum deviation.
To conclude, it is evident that hypISR can provide hard real-time in a regular Linux kernel, and since seL4 is not abundant software as Linux, hypISR can be used as a real-time solution in some cases.

C. Fast RPC

Here we demonstrate an RPC that eliminates context switches and therefore increases the remote call predictability. This section focuses on performance. We evaluated the round trip delay of calling a function that returns the time. For Xvisor, Native Linux and RT PREEMPT we used ptsematest, which is part of the Linux rt-test suite. Ptsematest measures interprocess latency communication with POSIX mutexes. Ptssematest starts two threads and synchronizes them with pthread_lock and pthread_unlock APIs. The receiving thread locks the mutex, and the sending thread releases the mutex. The time difference between the unlock to lock is the IPC duration.
In seL4 we used ptsematest-like test (sync.c) because ptsematest is not available in seL4. We used two threads, a consumer and a producer, the consumer waits on a semaphore (sync_bin_sem_t) and the producer signals the semaphore.
The hyplet test was a C program that made an RPC to a hyplet'ed process. The RPC returned the time stamp from the hyplet'ed process. The traveling time from the sender to the hyplet was recorded.
The reference test is to evaluate the cost of the function of the hyplet when not in HYP mode. We measured how much time it costs to call the function in the hyplet'ed process. Table III depicts the results. The tests were conducted on an idle system. The hyplet is the fastest RPC, even in the worst case. Xvisor

Name	Avg	Max
Ref	156ns	520ns
Hyplet	520ns	4.2μs
Normal	13μs	56μs
RT PRMT	15 μs	59μs
Xvisor	203μs	7067μs
seL4	8μs	17μs

TABLE III
ROUND TRIP RPC

results are the worst, it seems that its hypervisor preempts the OS for long durations. SeL4 RPC is on average is 13 times slower than the hyplet.
The maximum latency of the hyplet is may be due to the clock deviation, which is 140ppm. It is not cache misses or TLB EAT (Effective Access Time) because each exception level in ARM has its private cache and TLB, which is never evacuated.
Normal and RT PREEMPT results are close, which leads to the understanding that a context switch, on average, between two threads of the same process, on Linux in PI, is 14 μs compared to seL4's context switch which is on average 8 μs.

V. USABILITY

Due to the limitations of this paper, we do not present the full API. HypletUtil is a library that provides a services such as memory mappings to the nanovisor, synchronization primitives, events, printing and many others. Our library also provides **delicate mapping**. Delicate mapping is used when we want to map only certain global variables and functions to the nanovisor. For this, we use GCC sections. For example:

```
__attribute__ ((section("hyp"))) unsigned int a = 0;
unsigned int b = 0;
```

In this case, we want only to map the variable "a" and not "b". So, we grab the ELF (Executable Linkable Format) section "hyp" and map it to the nanovisor. For example, the below maps the section "hyp" to the nanovisor.
In non-inject mode, the hyplet can be removed the minute the process terminates, gracefully or not, or by explicitly unregisters the hyplet. Also, it is mandatory to lock the hyplet memory to the RAM to avoid relocation, invalidation, or swapping.

VI. RELATED WORK

The extended Berkely Packet Filter (eBPF) [5] is described as an in-kernel VM, and eBPF provides the ability to attach a program to a certain tracepoint in the kernel. Whenever the kernel reaches the tracepoint, the program is executed without a context switch. eBPF is undergoing massive development and is mainly used for packet inspection, tracing, and probing. It runs in kernel mode, which is considered unsafe, but it uses a verifier to check that there are no illegal accesses to kernel areas or some tampering registers. Access to the user-space is enabled through memory maps. Also, eBPF uses LLVM and requires clang to generate a JIT code and has a small instruction set. As a consequence, eBPF has serious limitations. Particularly, only a subset of the C language can be compiled into eBPF; as such eBPF has no loops, no native assembly, no static variables, and no atomics. Furthermore, using eBPF may not take a long time and is restricted to 4096 instructions. This is not the case with the hyplet. The hyplet is not a program that executes in the kernel's address space, but in the user's address space. So, there is no need for maps to share data between the user and the kernel. The hyplet does not require any special compiler extensions and is much less restricted (what mapped prematurely can be accessed) and less complicated to use compared to eBPF. In general, the hyplet is meant to process events in user-space while eBPF collects data and processes it in kernel mode.
Hyperupcalls [2], which are ePBF extensions for a hypervisor, are a means to run hypervisor code in the guest's kernel context. Hyperupcalls are intended mainly for monitoring the health of the guest VM and are available only for the x86 architecture. The hyplet, in contrast, only uses the hypervisor and is not intended for the control and management of VMs. Nevertheless, it is possible to combine eBPF and the hyplet technologies so an eBPF program invokes a hyplet directly.
There has been a significant amount of research on secure microkernels and microvisors. A prominent microvisor is the OKL4 by Open kernel labs. The OKL4 microvisor [8] is a secure hypervisor that is supported by Cog systems and General Dynamics. The OKL4 microvisor supports both paravirtualization and pure virtualization. It is designed for the IoT and mobile industries and supports ARMv5, ARMv6, ARMv7, and ARMv8. Unlike the hyplet, the OKL4 microvisor is a full kernel executing in HYP mode. OKL4 microvisor has an open source sister project microkernel called seL4. Installing seL4 and running it is a challenging task because seL4 requires expertise and the adoption of the hardcore of the code. Another L4 para-virtualization technology is the L^{4}Linux [7] para-virtualized Linux kernel, that runs on top of the L4Re [13] microkernel. This system demonstrates real-time when threads execute in the microkernel. It transparently migrates a Linux thread to an L4Re thread. This is possible since the L^{4}Linux reuses address spaces and threading APIs of the L4Re microkernel. [11] presents a hard real-time in x86 and ARM. However, this technology requires a special Linux variant kernel and an SMP machine. The hyplet was ported to several kernel versions, (3.18 (android), 4.4, 4.1, 4.10, and 4.17) and several SOMs (Mediatech phone, Hikey, or any other ARMv8a processors) and it can execute on a single processor. This is possible because we re-used many of Linux virtualization capabilities (KVM).
Dune [4] is a system that provides a process rather than a machine abstraction through virtualization. Dune offers a sandbox for untrusted code, a privilege separation facility, and a garbage collector. Dune is implemented on Intel architecture and can be implemented with hyplets. However, this implementation means coercing a VM on hyplets, which is not the intended use of a hyplet.
In the area of pure virtualization, some efforts, such as Jailhouse and Xvisor, were made to run a guest OS as an

RTOS. Jailhouse demonstrates that it is possible to run an RTOS guest on top of a thin hypervisor and still achieve low latencies.
In the Linux area, the topic of user-space drivers handling IO events and exists in the Linux kernel inside the Universal I/O (UIO) framework. The UIO device driver is a user-space driver that blocks until an interrupt arrives. UIO offers an easy way to interact with various hardware devices. However, UIO device drivers are not suitable for devices with a high interrupt frequency.

VII. Summary

A. Future work

We intend to implement the hyplet for PowerPC. PowerPC shares some ARM capabilities, and the results will determine whether using the hyplet is efficient. We expect that ARM virtualization host extension becomes available for commercial use, and test the VHE hyplet performance.

B. Conclusions

We have introduced a new way ARM hypervisor instructions can enhance Linux performance in real-time systems. These features allow for security and performance benefits. The hyplet allows coding interrupts with a predictable μs latency and highly efficient RPC. We have implemented hyplets variant as security solutions for ARM.

References

[1] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[2] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 97–112, 2018.

[3] Maxim Baryshnikov. Jailhouse hypervisor. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

[4] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

[5] Jonathan Corbet. Bpf comes to firewalls, 2018.

[6] Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. *ACM SIGOPS Operating Systems Review*, 31(5):66–77, 1997.

[8] 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*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[9] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 1, pages 144–151. IEEE, 2008.

[10] "Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood". sel4: formal verification of an os kernel. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

[11] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Predictable low-latency interrupt response with general-purpose systems. In *Proceedings of OSPERT2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications OSPERT 2017*, pages 19–24, 2017.

[12] William Rosenblatt, Stephen Mooney, and William Trippe. *Digital rights management: business and technology*. John Wiley & Sons, Inc., 2001.

[13] Alexander Warg and Adam Lackorzynski. The fiasco. oc kernel and the l4 runtime environment (l4re). avail.

[14] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 277–280. IEEE, 2016.

[15] Raz Ben Yehuda and Nezer Jacob Zaidenberg. Protection against reverse engineering in arm. *International Journal of Information Security*, 19(1):39–51, 2020.

[16] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan. Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 421–426. IEEE, 2010.

[17] Baojing Zuo, Kai Chen, Alei Liang, Haibing Guan, Jun Zhang, Ruhui Ma, and Hongbo Yang. Performance tuning towards a kvm-based low latency virtualization system. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1–4. IEEE, 2010.

[18] Op tee , linaro limited: Open portable trusted execution environment

PVII

HYPERVISOR MEMORY INTROSPECTION AND HYPERVISOR BASED MALWARE HONEYPOT.

by

Nezer Zaidenberg, Michael Kiperberg, Raz Ben Yehuda, Roee Leon, Asaf Alagawi and Amit Resh. 2020

Information Systems Security and Privacy. ICISSP 2019. Communications in Computer and Information Science, vol 1221

Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot

Nezer Jacob Zaidenberg[1,2,5(✉)], Michael Kiperberg[3,5], Raz Ben Yehuda[2,5], Roee Leon[2,5], Asaf Algawi[2,5], and Amit Resh[4,5]

[1] College of Management Academic Studies, Rishon LeZion, Israel
[2] University of Jyväskylä, Jyväskylä, Finland
[3] SCE, Ashdod, Israel
[4] Shenkar College, Ramat Gan, Israel
[5] TrulyProtect Oy, Tel Aviv, Israel
{nezer,michael,raz,roee,asaf,amit}@trulyprotect.com

Abstract. Memory acquisition is a tool used in advanced forensics and malware analysis. Various methods of memory acquisition exist. Such solutions are ranging from tools based on dedicated hardware to software-only solutions. We proposed a hypervisor based memory acquisition tool. [22]. Our method supports ASLR and Modern operating systems which is an innovation compared to past methods [27,36]. We extend the hypervisor assisted memory acquisition by adding mass storage device honeypots for the malware to cross and propose hiding the hypervisor using bluepill technology.

Keywords: Live forensics · Memory forensics · Memory acquisition · Virtualization · Reliability · Atomicity · Integrity of a memory snapshot · Forensic soundness

1 Introduction

Nowadays, cyber-attacks are so sophisticated that it is almost impossible to perform static analysis on them. Many of the recent attacks have means to detect debuggers and similar dynamic analysis tools. Upon detection of an inspection, the malicious software deviates from its normal behaviour, thus rendering the analysis useless. The malicious software will only activate its destructive parts if no inspection tool is running.

Furthermore, we are often interested in a forensic analysis of such an attack. In some case (such as Wannacry virus [31]) such analysis may locate weakness in the attack. In others (such as Stuxnet [24]) such analysis performed in the post mortem may inform us about the preparators of the attack. Also, in defence oriented cases, the origin of the attack may be another nation. In such cases it is required to know which nation have performed the attack. Furthermore, sometimes false feedback can be provided to the sender which is also required. Since dynamic analysis is doomed to failure, the forensic analysis of is divided into two steps

P. Mori et al. (Eds.): ICISSP 2019, CCIS 1221, pp. 317–334, 2020.
https://doi.org/10.1007/978-3-030-49443-8_15

Memory Acquisition. An acquisition tool (such as [27,36,37,52]) acquires the contents of the system memory (RAM). The tool stores the memory on some file for offline analysis.

Static Analysis. An analysis tool (such as rekall [10] or volatility [26]) is used on the file that was acquired in the previous step. The analysis tools are searching for malware and other anomalies.

During the operating system running time, the operating system updates its data structures and pointers. If the user performs, the memory acquisition while the system is running, the user may acquire an inconsistent image of the system memory. For example we examine the common task of creating a new task. While creating a new task, the operating system performs allocation for the new task memory in one memory region and also updates the task table in another memory region.

If the process table is dumped on one chunk and the process itself is dumped on another chunk them an inconsistency is likely to appear. For example, a task in the task table that does not point to memory or task that exists in memory but does not exist on the task table.

Memory inconsistencies look like anomaly and are likely to confuse the detection process. Therefore, preventing measures must be taken to avoid inconsistencies in the acquired memory image. We present a software hypervisor-based tool for consistent memory acquisition, [22]. The Hypervisor can also be used to create tripwires to detect malicious software. We added this feature to the current work. Furthermore, inspected malicious software can alter its behavior when inspected. Therefore we added blue pill technology to the hypervisor to enable the hypervisor to introspect the system and grab the system memory without being detected.

We use the hypervisor's ability to configure access rights of memory pages to solve the problem of inconsistencies as follows:

1. When memory acquisition is started, the hypervisor configures all memory pages to be non-writable.
2. When any process attempts to write to any memory page (hereby P), the hypervisor is notified.
3. The hypervisor copies the contents of P to an internal buffer and configures P to be writable.
4. The hypervisor performs the dump emptying its internal buffer as first priority. If no data remains to be copied in the internal buffer, then the hypervisor sends other pages and configures them to be writable.
5. Writable pages no longer triggers event in the hypervisor.
6. The hypervisor also serves as an honeypot. The hypervisor is now looking for attacks. The hypervisor is now seducing attacks as some sort of honeypot [32].

Steps 1 through 4 are described in multiple previous works [27,36].
Three problems arise with the described method in modern systems.

Multi-cores. Each processor core has direct access to the main memory. Thus each core can modify any memory page. Therefore, when the hypervisor starts memory acquisition, it must configure all memory pages, *onallprocessors* to be non-writable.

Delay Sensitive Pages. Generally, interrupt service routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received in a timely manner; therefore, the registration of an interrupt occurrence, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable.

ASLR. Address space layout randomization [45], a security feature employed by modern operating systems, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. With ASLR, the location of the delay-sensitive pages is not known in advance.

We solved the problems mentioned above in [22] in the following method. Our hypervisor invokes an operating system's mechanism to perform an atomic access rights configuration on all the processors. Section 4.3 describes the invocation process, which allows our hypervisor to call an operating system's function in a safe and predictable manner.

We solved the delay sensitivity problem by copying the delay-sensitive pages to the hypervisor's internal buffer in advance, i.e., when the hypervisor starts memory acquisition.

We solved the ASLR complication by inspecting the operating system's memory regions map. Thu we have showed how the location of the dynamic locations of the delay-sensitive pages is obtained. Section 4.2 contains a detailed description of ASLR handling in Windows 10 and our solution of the delay sensitivity problem in windows.

The contribution of our work is:

- We showed memory acquisition technique on systems with multiple processors.
- We showed a solution for the delay sensitive data
- We explained how the locations of sensitive pages can be obtained dynamically on Windows 10.

Furthermore, the malware can detect our memory acquisition hypervisor, proposed in [22]. Modern operating systems such as Windows and OSX are using hypervisors as part of the system, however the malware may also detect the hypervisor, suspect an inspection [51] and alter its behaviour.

Therefore, we have add two more contributions to this work

- Summary of recent work on building stealth hypervisors (blue pills) [1].
- we add new feature adding honey pot (trip wire) that will look like an interesting target and attack the malicious code to reveal itself and attack it [4].

2 Related Work

Windows versions before Windows server 2003sp1 contained a special device,

`\\\\Device\\PhysicalMemory`

This device map the entire physical memory. This device could be used to acquire the entire physical memory. However, Microsoft removed this device from modern windows operating systems [29]. This device can be used by malware as means to disrupt the memory acquisition process [9]. Furthermore, this method is not available in recent versions of Windows windows.

One can also use dedicated hardware for memory acquisition. A generic FireWire adapter can be used to acquire memory remotely [52]. A dedicated PCI card, named Tribble, is another option [9]. As well as any RDMA hardware. The advantage of a hardware solution is the ability of a PCI card to communicate with the memory controller directly, thus providing a reliable result even if the operating system itself was compromised. However, hardware-based solutions have three deficiencies:

1. Hardware based solutions require dedicated hardware thus increasing the cost of the implementation
2. Hardware based solution may be faster then software based solution but still do not provide atomic memory dump.
3. Microsoft's device guard [11] prevents using these tools.

Device Guard is a security feature recently introduced in Windows 10. Device guard utilizes IOMMU ([3]; [50]) to prevent malicious access to memory from physical devices [8]. When Device Guard is running, the operating system assigns each device a memory region that it is allowed to access. The DMA controller prevents any attempt to access memory outside this region, including memory acquisition. Recently, researchers proposed several hypervisor-based methods of memory acquisition. HyperSleuth [27] is a driver with an embedded hypervisor. Hyper-Sleuth hypervisor is capable of performing atomic and lazy memory acquisition. Lazy in terms of memory acquisition is the ability of the system to run normally while the memory is acquired. ForenVisor [36] is a similar hypervisor with also act as key logger and monitors hard-drive activity. Both HyperSleuth and ForenVisor works on Windows XP SP3 with only one processor enabled. We show how the idea of HyperSleuth and ForenVisor can be adapted to multi-processor systems executing Windows 10.

3 Background

3.1 Hypervisors

Our primary system component is the hypervisor. We describe the hypervisor design in Sect. 4. We distinguish between two families of hypervisors: full hypervisors and thin hypervisors. Full hypervisors like Xen [2], VMware Workstation

[47], and Oracle VirtualBox [33]. Can run several operating systems in parallel. Almost all hypervisors, including ours and all the hypervisors aforementioned above, use hardware assisted virtualization. Hardware assisted virtualization is an instruct set providing efficient API to run multiple virtual machines. Intel use the term VT-x for their hardware assisted virtualization implementation. Running multiple operating systems efficiently is the primary goal of VT-x [18]. In contrast, thin hypervisors execute only a single operating system. The primary purpose of thin hypervisors is to enrich the functionality of an operating system. The main benefit of a hypervisor over kernel modules (device drivers) is the hypervisor's ability to provide an isolated environment, unlike containers (such as docker [7], kubernetes [16]). Thin hypervisors in the industry have various purposes including for real-time [5,15] and other purposes. We focus on thin-hypervisors use for security. Thin hypervisor can measure operating system's integrity validation [14,43], reverse engineering prevention [6,20] remote attestation [19,21]; malicious code execution prevention [25,38], in-memory secret protection [39], hard drive encryption [44], and memory acquisition [36].

Thin hypervisors perform fewer functions than full hypervisors; therefore, thin hypervisors are smaller than full hypervisors. Thus thin hypervisors are superior in their performance, security, and reliability. Our memory acquisition hypervisor is a thin hypervisor that is capable of acquiring a memory image of an executing system atomically. The hypervisor was written from scratch to achieve optimal performance. Similarly to an operating system, a hypervisor does not execute voluntarily but responds to events, e.g., execution of special instructions, generation of exceptions, access to memory locations, etc. The hypervisor can configure interception of (almost) each event. Trapping an event (a VM-exit) is similar to the handling of an interrupt, i.e., a predefined function executes when an event occurs. Another similarity with a full operating system is the hypervisor's ability to configure the access rights to each memory page through second-level address translation tables (SLAT tables) structure. Intel uses the name EPT for their SLAT implementation and use the terms interchangeably. An attempt to write to a non-writable (according to EPT) page induces a VM-exit and allows the hypervisor to act.

3.2 Lazy Hypervisor Memory Acquisition

Both HyperSleuth and ForenVisor are thin hypervisors and can be summarized as follows. The hypervisor is idle until it receives a memory acquisition command. After the command is received, the hypervisor configures the EPT to make all memory pages non-writable. An attempt to write to a page P will trigger a VM-exit, thus allowing the hypervisor to react. The hypervisor reacts by copying P to an internal buffer and making P writable again. Thus, Future attempts to write to P will not trigger a VM-exit. The hypervisor sends the buffered pages to a remote machine via a communication channel. The required buffer size depends on the communication channel bandwidth and the volume of modified pages. If the communication channel allows sending more data than is the buffer

contains, the hypervisor sends other non-writable pages and configures them to be writable. This process continues until all pages are writable.

3.3 Delay-Sensitive Pages and ASLR

We examine Interrupt Service Routines (ISR). Generally, ISR routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was handled. The acknowledgement must be received in real-time; therefore, the registration of an interrupt, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable. The authors of HyperSleuth and ForenVisor did not address this issue. We assume that this problem did not occur on single-core Windows XP SP3, which previous works used. Address space layout randomization, a security feature employed by the modern operating system, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. This behavior is useful against a wide range of attacks [12] because the address of potentially vulnerable modules is not known in advance. However, for the same reason, the address of the delay-sensitive pages is also unpredictable.

3.4 Honeypots

Honeypots in the network case have been researched for a long time [4]. A Honeypot is a network device that appears vulnerable to an attack. The malicious software reveals itself by attacking the honeypot. The system administrator monitors the honeypot, detect the attack and remove the attacker.

Honeypot (and Anti-Honeypots [23]) are well researched [28] in the network case. Network honeypots are also used in forensics [34].

Our honeypot differs from prior art as we use the hypervisors to create honeypot devices within the machine instead of the network. This innovation allows us to trigger memory collection just as the malware start operating.

4 Design

4.1 Initialization

We implemented the hypervisor as a UEFI application [46]. The UEFI application loads before the operating system, allocates all the required memory, and initializes the hypervisor. After initialization, the UEFI application terminates, thus allowing the operating system boot loader to initialize the operating system. We note that while the UEFI application that started the hypervisor is terminated, the hypervisor remains active.

In order to protect itself from a potentially malicious environment, the hypervisor configures the SLAT such that any access to the code and the data of the

hypervisor is prohibited. With this exception, the SLAT is configured to be an identity mapping that allows full access to all the memory pages (Fig. 1).

The hypervisor remains idle until an external event triggers its memory acquisition functionality. The external event might be the reception of a network packet, insertion of a USB device, starting a process, invocation of a system call, etc. In our prototype implementation, we used a special `CPUID` instruction, which we call `FREEZE`, as a trigger. `CPUID` is an Intel assembler instruction that the hypervisor must trap (preform VM-Exit) according to the architecture specification.

When receiving `FREEZE`, the hypervisor performs two actions:

1. Identifies and copies the delay-sensitive pages.
2. configure the access rights of all memory pages on all cores to be non-writable.

After all pages are marked non-writable the hypervisor reacts to page modification attempts by making the page writable and copying it to an internal buffer. (similar to ForenVisor and Hyper Sleuth) The hypervisor exports the pages stored in the internal buffer in response to another special `CPUID` instructions, which we call `DUMP`. (or another trigger as above)

If the queue is not full, then the hypervisor exports other non-writable pages and marks the exported pages writable.

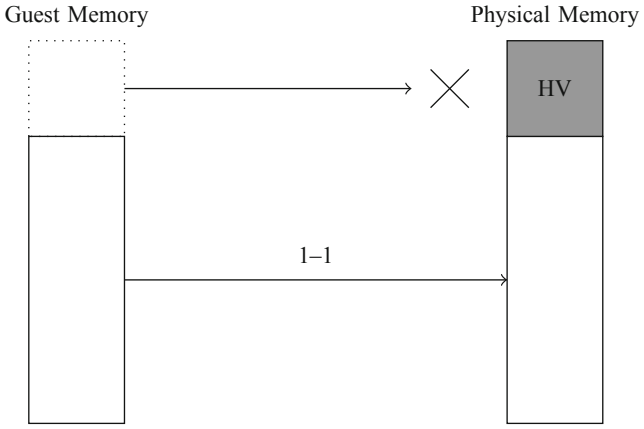


Fig. 1. Mapping between the physical address space as observed by the operating system (left) and the actual physical address space. The mapping is an identity mapping with the exception of the hypervisor's pages, which are not mapped at all. Originally appeared at [22].

Algorithm 1: Memory Acquisition. Originally appeared at [22].

```
1: file ← Open(...)
2: FREEZE()
3: while DUMP(addr, page) do
4:     Seek(file, addr)
5:     Write(file, page)
6: Close(file)
```

Algorithm 1 describes how `FREEZE` and `DUMP` can be used to acquire an atomic image of the memory. First, the algorithm opens a file that will contain the resulting memory image. Then, `FREEZE` is invoked, followed by a series of `DUMP`s. When the `DUMP` request returns *false*, the file is closed and the algorithm terminates.

The file now contain the entire system memory and can be analyzed by other tools such as volatility [26].

4.2 Delay

We explained in Sect. 3.3 that certain pages cannot not be configured as non-writable. Moreover, due to ASLR, the hypervisor has to discover the location of these pages at run time. The discover process and the Windows 10 data structures that are used are described here.

Table 1. Windows ASLR-related data structures. Originally appeared at [22].

Offset	Field/Variable Name	Type
X	System Call Service Routine	*Code*
...	...	...
+0xFB100	MiState	MI_SYSTEM_INFORMATION
+0x1440	Vs	MI_VISIBLE_STATE
+0x0B50	SystemVaRegions	MI_SYSTEM_VA_ASSIGNMENT[14]
+0x0000	[0]	MI_SYSTEM_VA_ASSIGNMENT
+0x0000	BaseAddress	uint64_t
+0x0008	NumberOfBytes	uint64_t

Windows 10 defines a global variable `MiState` of type `MI_SYSTEM_INFORMATION`. The hypervisor can easily locate this variable as it has a constant offset from the system call service routine, whose address is stored in the LSTAR register (Table 1). The `MI_SYSTEM_INFORMATION` structure has a field named `Vs` of type `MI_VISIBLE_STATE`. Finally, the `MI_VISIBLE_STATE` structure has a field named `SystemVaRegions`, which is an array of 15 pairs. Each pair corresponds to a memory region whose address was chosen at random during the operating system's initialization. The first element of the pair is the random address and the second element is the region's size. A description of each memory region is given in Table 2. A more detailed discussion of the memory regions appears in [40]. Our empirical study shows that the following regions contain delay-sensitive pages:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged
4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Therefore, the hypervisor never makes these regions non-writable.

Table 2. Memory Regions. Originally appeared at [22].

Index	Name
0	MiVaUnused
1	MiVaSessionSpace
2	MiVaProcessSpace
3	MiVaBootLoaded
4	MiVaPfnDatabase
5	MiVaNonPagedPool
6	MiVaPagedPool
7	MiVaSpecialPoolPaged
8	MiVaSystemCache
9	MiVaSystemPtes
10	MiVaHal
11	MiVaSessionGlobalSpace
12	MiVaDriverImages
13	MiVaSystemPtesLarge

4.3 Multicore

The hypervisor responds to `FREEZE`, a memory acquisition request, by copying the delay-sensitive pages to an inner queue and configuring all other pages to be non-writable. However, when multiple processors are active, the access rights configuration must be performed atomically on all processors.

Operating systems usually use inter-processor interrupts (IPIs) [18] for synchronization between processors. It seems tempting to use IPIs also in the hypervisor, i.e., the processor that received `FREEZE` can send IPIs to other processors, thus requesting them to configure the access rights appropriately. Unfortunately, this method requires the hypervisor to replace the operating system's interrupt-descriptors table (IDT) with the hypervisor's IDT. This approach has two deficiencies:

1. Kernel Patch Protection (KPP) [13], a security feature introduced by Microsoft in Windows 2003, performs a periodic validation of critical kernel structures in order to prevent their illegal modification. Therefore, replacing the IDT requires also intercepting KPP's validation attempts, which can degrade the overall system performance.
2. Intel processors assign priorities to interrupt vectors. Interrupts of lower priority are blocked while an interrupt of a higher priority is delivered. Therefore, the hypervisor cannot guarantee that a sent IPI will be handled within a predefined time. Suspending the operating system for long periods can cause the operating system's watchdog timer to trigger a stop error (BSoD).

We present a different method to solve the inter-processor synchronization problem that is based on a documented functionality of the operating system itself. The `KeIpiGenericCall` function [30] receives a callback function as a parameter and executes it on all the active processors simultaneously. We propose to use the `KeIpiGenericCall` function to configure the access rights simultaneously on all the processors.

Because it is impossible to call an operating system function from within the context of the hypervisor, the hypervisor calls the `KeIpiGenericCall` function from the context of the (guest) operating system. In order to achieve this, the hypervisor performs several preparations and then resumes the execution of the operating system. Algorithm 2 presents three functions that together perform simultaneous access rights configuration on all the active processors. The first function, HandleCpuid, is part of the hypervisor. This function is called whenever the operating system invokes a special `CPUID` instruction. Two other functions, GuestEntry and Callback, are mapped by the hypervisor to a non-occupied region of the operating system's memory.

Algorithm 1 begins with a special `CPUID` instruction, called `FREEZE`. This instruction is handled by lines 2–5 in Algorithm 2: the hypervisor maps GuestEntry and Callback, saves the current registers' values and sets the instruction pointer to the address of GuestEntry. The GuestEntry function calls the operating system's KeIpiGenericCall, which will execute Callback on all the active processors. The Callback function performs another special `CPUID` instruction, called `CONFIGURE`, which causes the hypervisor to configure the access rights of all (but the delay-sensitive) memory pages on all the processors. This is handled by lines 6–7 of the algorithm, where we omitted the

Algorithm 2: Simultaneous access rights configuration on all the active processors. Originally appeared at [22].

```
 1: function HandleCpuid(reason)
 2:     if reason=FREEZE then
 3:         Map GuestEntry and Callback
 4:         Save registers
 5:         RIP ← GuestEntry
 6:     else if reason=CONFIGURE then
 7:         ...
 8:     else if reason=RESUME_OS then
 9:         Restore registers
10:     else if reason=DUMP then
11:         ...
12:     ...
13: function GuestEntry
14:     KeIpiGenericCall(Callback)
15:     CPUID(RESUME_OS)
16: function Callback
17:     CPUID(CONFIGURE)
```

configuration procedure itself. After the termination of the CALLBACK function, the control returns to the GUESTENTRY function, which executes a special `CPUID` instruction, named `RESUME_OS`. In response, the hypervisor restores the registers' values, which were previously saved in line 4. The operation continues from the instruction following `FREEZE`, which triggered this sequence of events.

5 Honeypot

We augmented the project by creating an honeypot. We create a virtual mass storage device. We assume malware such as stuxnet [24] or computer viruses attempt to detect mass storage device for replication purposes.

We use special device detect device access to the virtual device. In some cases such device access can serve as `FREEZE` trigger.

The honeypot is made of two main components

Access Identification Component for Storage Components. The component is installed as a Kernel module and extends to a virtual disk operating system that identifies as a physical component and monitors access to it. When a process accesses the storage component, the process data enters the queue of the malicious processes that are waiting for the action of the enforcement component.

Enforcement Component. This component is managed in the user space of the operating system and is waiting to be called, when it is called it invokes the `FREEZE` and `DUMP` commands.

6 Blue-Pilling

[41] introduced the blue pill. It was with the concept of the blue pill and red pill that the virtualization concept became so closely related to cybersecurity. The blue pill is a rootkit that takes control of the victim's computer [42]. The blue pill is very hard to detect. Since the blue pill is an hypervisor it is not visible on the standard task manager or even works in the same address space as the operating system. Since our interest is in acquiring reliable memory image that includes any malicious software and since malicious software may hide their presence if a memory acquisition takes place we recommend that blue pill technology will be added to the hypervisor. Rotkowska also coined the term red pill. The red pill is meant to detect and counter the blue pill. The red pill is a hardware or software tool that is designed to detect such malicious camouflaged hypervisor-based rootkit.

Some users start a virtual machine as a sandbox to detect malicious code in contained environment. When using suspected code they install it first on the virtual machine on only if no attacks were spotted they move the software to their physical machine.

To counter the above routine, some malware use simple red pills to detect hypervisors. These malware will not use their offensive features if an hypervisor is present. Therefore, it is vital for the memory acquisition hypervisor to also act as a blue-pill stealth hypervisor.

[1] describes the current status of blue pill hypervisors. We recommend that these methods will be added to this solution as well.

7 Evaluation

We evaluate the performance of the hypervisor and its memory usage. First, we demonstrate the overall performance impact of the hypervisor. (We compare the performance of a normal system to a system with a hypervisor that does nothing)

Next, we analyze the memory usage of the hypervisor. Finally, we evaluate the performance of the memory acquisition process.

All the experiments were performed in the following environment:

- CPU: Intel Core i5-6500 CPU 3.20 GHz (4 physical cores)
- RAM: 16.00 GB
- OS: Windows 10 Pro x86-64 Version 1803 (OS Build 17134.407)
- C/C++ Compiler: Microsoft C/C++ Optimizing Compiler Version 19.00.23026 for x86.

7.1 Performance Impact

We demonstrate the performance impact of the hypervisor on the operating system. We picked two benchmarking tools for Windows:

1. PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457 64 , SystemInfo – 5.4.642, PCMark 10 System 1.0.1457,
2. Novabench. Version Info: 4.0.3 – November 2017.

Each tool performs several tests and displays a score for each test. We invoked each tool twice: with and without the hypervisor. The results of PCMark, and Novabench are depicted in Figs. 2 and 3, respectively. We can see that the performance penalty of the hypervisor is approximately 5% on average. This figure is equivalent to similar results achieved by top vendors [17,48,49].

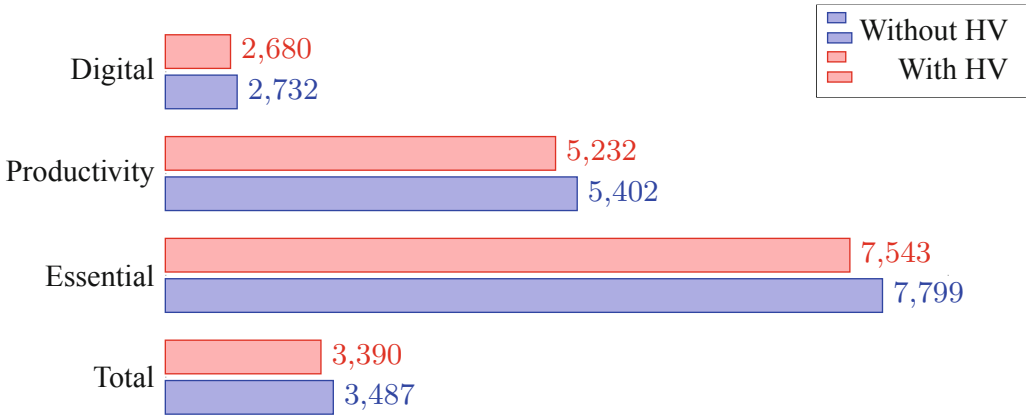


Fig. 2. Scores (larger is better) reported by PCMark in four categories: Digital Content Creation, Productivity, Essential, and Total. Originally appeared at [22].

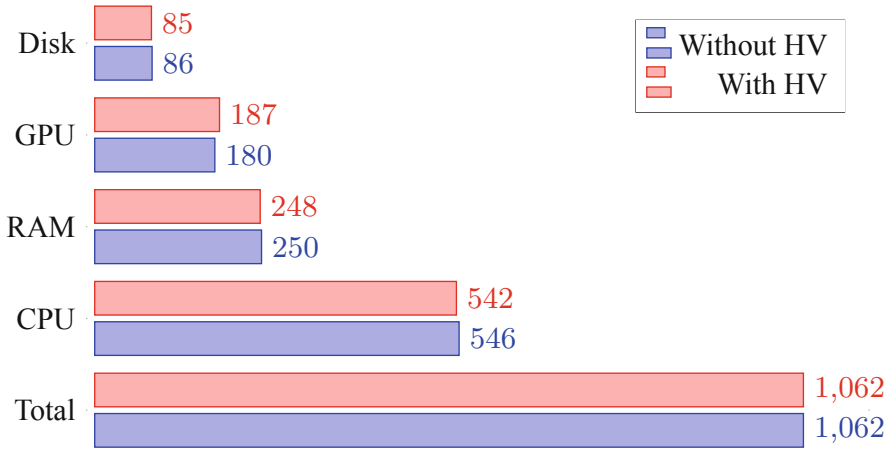


Fig. 3. Scores (larger is better) reported by Novabench in five categories: Disk, GPU, RAM, CPU, and Total. Originally appeared at [22].

7.2 Memory Usage

The memory used by the hypervisor can be divided into three main parts:

1. the code and the data structures of the hypervisor,
2. the EPT tables used to configure the access rights to the memory pages, and
3. the queue used to accumulate the modified pages.

Figure 4 presents the memory usage of the hypervisor including its division.

The size of the queue is mainly dictated by the number of delay-sensitive pages. Table 3 presents the typical size of each memory region.

Pages belonging to the following regions are copied by the hypervisor:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged

4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Their total size is ≈60 MB. The size of the queue should be slightly larger than the total size of the delay-sensitive pages because regular pages can be modified by the operating system before the content of the queue is exported. Our empirical study shows that it is sufficient to enlarge the queue by 60 MB.

Table 3. Memory Regions' Sizes. Originally appeared at [22].

Index	Name	Size (MB)
0	MiVaUnused	6
1	MiVaSessionSpace	100
2	MiVaProcessSpace	0
3	MiVaBootLoaded	0
4	MiVaPfnDatabase	0
5	MiVaNonPagedPool	6
6	MiVaPagedPool	0
7	MiVaSpecialPoolPaged	5
8	MiVaSystemCache	52
9	MiVaSystemPtes	0
10	MiVaHal	0
11	MiVaSessionGlobalSpace	0
12	MiVaDriverImages	8

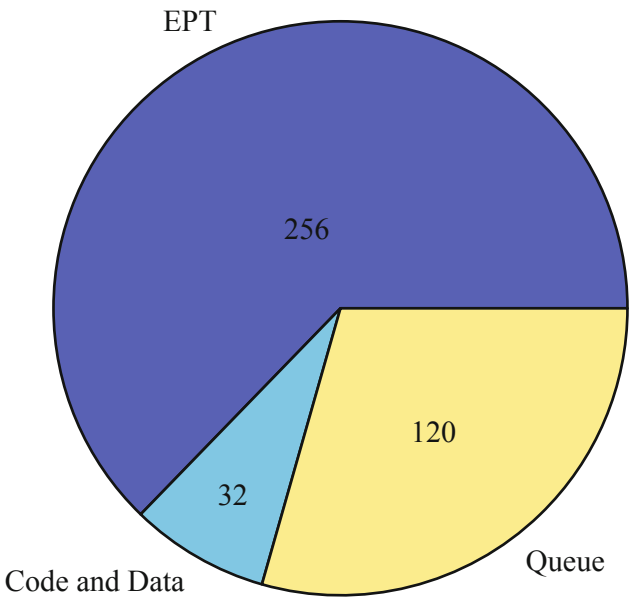


Fig. 4. Hypervisor's Memory Usage [MB]. Originally appeared at [22].

7.3 Memory Acquisition Performance

We examine the correlation between the speed of memory acquisition and the overall system performance. Figure 5 shows the results. The horizontal axis represents the memory acquisition speed. The maximal speed we could achieve was 97920KB/s. At this speed, the system became unresponsive and the benchmarking tools failed. The vertical axis represents the performance degradation (in percent) measured by PCMark and Novabench. More precisely, denote by $t_i(x)$ the *Total* result of benchmark $i = 1, 2$ (for PCMark and Novabench, respectively) with acquisition speed of x; then, the performance degradation $d_i(x)$ is given by

$$d_i(x) = 1 - \frac{t_i(x)}{t_i(0)}. \tag{1}$$

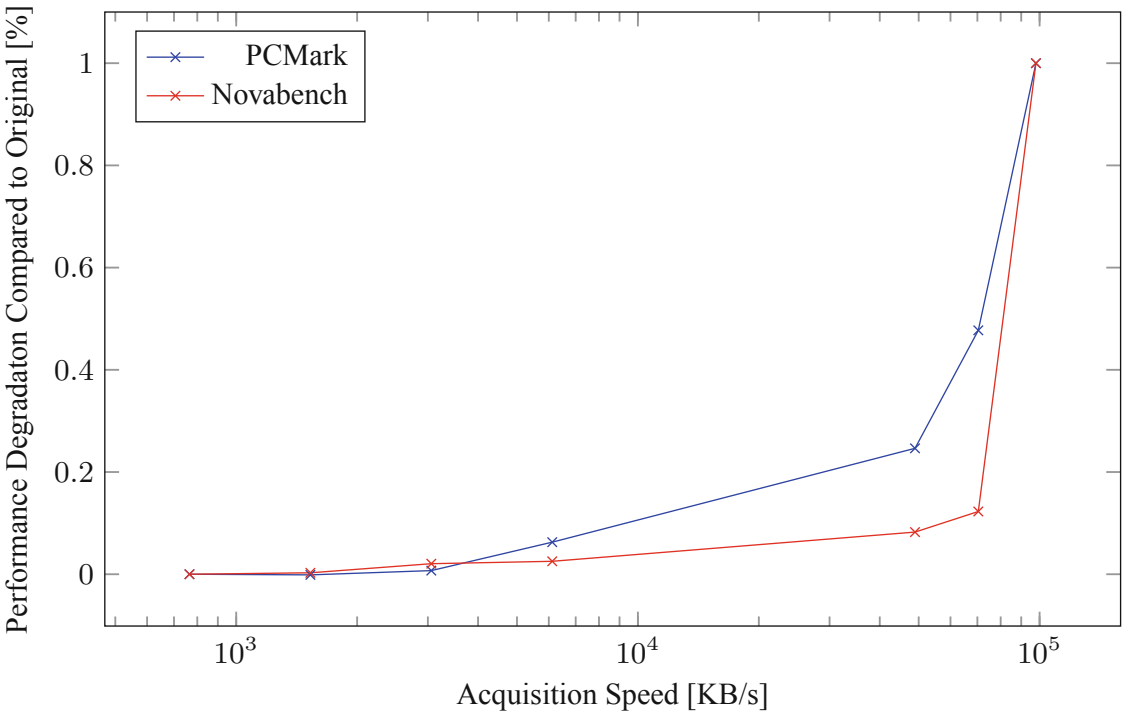


Fig. 5. Performance degradation due to memory acquisition. Originally appeared at [22].

8 Conclusion

Our method is a small improvement over previously described methods with similar design. We describe five improvements over the previously described methods:

1. Our hypervisor supports multiple cores and processors.
2. Our hypervisor supports modern operating systems, e.g., Windows 10 and Linux.

3. Our hypervisor includes honeypot for malicious hypervisors.
4. We have proposed means to use blue-pill techniques so that malware will can not detect the hypervisor.
5. Additionally, our new paper also includes a USB honeypot as trip wire that does not appear in [22].

References

1. Algawi, A., Kiperberg, M., Leon, R., Resh, A., Zaidenberg, N.: Creating modern blue pills and red pills. In: European Conference on Cyber Warfare and Security, pp. 6–14. Academic Conferences International Limited, July 2019
2. Barham, P., et al.: Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review, vol. 37, pp. 164–177. ACM (2003)
3. Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., Van Doorn, L.: The price of safety: evaluating iommu performance. In: The Ottawa Linux Symposium, pp. 9–20 (2007)
4. Ben Yehuda, R., Kevorkian, D., Zamir, G.L., Walter, M.Y., Levy, L.: Virtual USB honeypot. In: Proceedings of the 12th ACM International Conference on Systems and Storage, pp. 181–181, May 2019
5. Ben Yehuda, R., Zaidenberg, N.: Hyplets-multi exception level kernel towards Linux RTOS. In: Proceedings of the 11th ACM International Systems and Storage Conference, pp. 116–117, June 2018
6. Ben Yehuda, R., Zaidenberg, N.J.: Protection against reverse engineering in ARM. Int. J. Inf. Secur. **19**(1), 39–51 (2019). https://doi.org/10.1007/s10207-019-00450-1
7. Boettiger, C.: An introduction to Docker for reproducible research. ACM SIGOPS Oper. Syst. Rev. **49**(1), 71–79 (2015)
8. Brendmo, H.K.: Live forensics on the windows 10 secure kernel. Master's thesis, NTNU (2017)
9. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Dig. Invest. **1**(1), 50–60 (2004)
10. Cohen, M.: Rekall Memory Forensics Framework. DFIR Prague (2014)
11. Durve, R., Bouridane, A.: Windows 10 security hardening using device guard whitelisting and applocker blacklisting. In: 2017 Seventh International Conference on Emerging Security Technologies (EST), pp. 56–61. IEEE (2017)
12. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over ASLR: attacking branch predictors to bypass ASLR. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 40. IEEE Press (2016)
13. Field, S.: An introduction to kernel patch protection (2006). http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx
14. Franklin, J., Seshadri, A., Qu, N., Chaki, S., Datta, A.: Attacking, repairing, and verifying SecVisor: a retrospective on the security of a hypervisor. Technical report CMU-CyLab-08-008, Carnegie Mellon University (2008)
15. Heiser, G., Leslie, B.: The OKL4 Microvisor: convergence point of microkernels and hypervisors. In: Proceedings of the first ACM ASIA-pacific Workshop on Systems, pp. 19–24. ACM, August 2010
16. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media Inc., Sebastopol (2017)

17. Huber, N., von Quast, M., Brosig, F., Kounev, S.: Analysis of the performance-influencing factors of virtualization platforms. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6427, pp. 811–828. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16949-6_10
18. Intel Corporation: Intel© 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (2018)
19. Kiperberg, M., Resh, A., Zaidenberg, N.J.: Remote attestation of software and execution- environment in modern machines. In: 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing (CSCloud), pp. 335–341. IEEE (2015)
20. Kiperberg, M., Leon, R., Resh, A., Algawi, A., Zaidenberg, N.J.: Hypervisor-based protection of code. IEEE Trans. Inf. Forensics Secur. **14**(8), 2203–2216 (2019)
21. Kiperberg, M., Zaidenberg, N.: Efficient remote authentication. In: Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013, p. 144. Academic Conferences Limited (2013)
22. Kiperberg, M., Leon, R., Resh, A., Algawi, A., Zaidenberg, N.:. Hypervisor-assisted atomic memory acquisition in modern systems. In: Mori, P., Furnell, S., Camp, O. (eds.), ICISSP 2019: Proceedings of the 5th International Conference on Information Systems Security and Privacy, vol. 1, pp. 155–162 (2019)
23. Krawetz, N.: Anti-honeypot technology. IEEE Secur. Privacy **2**(1), 76–79 (2004)
24. Langner, R.: Stuxnet: dissecting a cyberwarfare weapon. IEEE Securi. Privacy **9**(3), 49–51 (2011)
25. Leon, R., Kiperberg, M., Zabag Leon, A.A., Resh, A., Algawi, A., Zaidenberg, N.J.: Hypervisor-based whitelisting of executables. IEEE Securi. Privacy (2019)
26. Macht, H.: Live memory forensics on android with volatility. Friedrich-Alexander University Erlangen-Nuremberg (2013)
27. Martignoni, L., Fattori, A., Paleari, R., Cavallaro, L.: Live and trustworthy forensic analysis of commodity production systems. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 297–316. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15512-3_16
28. Mairh, A., Barik, D., Verma, K., Jena, D.: Honeypot in network security: a survey. In: Proceedings of the 2011 International Conference on Communication, Computing & Security, pp. 600–605. ACM, February 2011
29. Microsoft Corporation (2009). Device /PhysicalMemory Object. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc787565(v=ws.10). Accessed 02 Nov 2018
30. Microsoft Corporation (2018). KeIpiGenericCall function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keipigenericcall
31. Mohurle, S., Patil, M.: A brief study of wannacry threat: ransomware attack 2017. Int. J. Adv. Res. Comput. Sci. **8**(5) (2017)
32. Moore, C.:. Detecting ransomware with honeypot techniques. In: 2016 Cybersecurity and Cyberforensics Conference (CCC), pp. 77–81. IEEE, August 2016
33. Oracle (2018). VirtualBox. https://www.virtualbox.org/
34. Pouget, F., Dacier, M.: Honeypot-based forensics. In: AusCERT Asia Pacific Information Technology Security Conference, May 2004
35. Provos, N.: A virtual honeypot framework. In: USENIX Security Symposium, vol. 173, no. 2004, pp. 1–14, August 2004
36. Qi, Z., Xiang, C., Ma, R., Li, J., Guan, H., Wei, D.S.: Forenvisor: a tool for acquiring and preserving reliable data in cloud live forensics. IEEE Trans. Cloud Comput. **5**(3), 443–456 (2017)

37. Reina, A., Fattori, A., Pagani, F., Cavallaro, L., Bruschi, D.: When hardware meets software: a bulletproof solution to forensic memory acquisition. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 79–88. ACM (2012)
38. Resh, A., Kiperberg, M., Leon, R., Zaidenberg, N.J.: Preventing execution of unauthorized native code software. Int. Dig. Content Technol. Appl. **11** (2017)
39. Resh, A., Zaidenberg, N.: Can keys be hidden inside the CPU on modern windows host. In: Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013, p. 231. Academic Conferences Limited (2013)
40. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows Internals. Pearson Education, London (2012)
41. Rutkowska, J.: Introducing blue pill. The official blog of the invisiblethings. org, 22, 23. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html (2006)
42. Rutkowska, J.: Subverting VistaTM Kernel for Fun and Profit. Black Hat Briefings, Singapore (2006)
43. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSES. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 335–350. ACM (2007)
44. Shinagawa, T., et al.: Bitvisor: A thin hypervisor for enforcing I/O device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2009, New York, NY, USA, pp. 121–130. ACM (2009)
45. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE Symposium on Security and Privacy, pp. 574–588. IEEE. Chicago, May 2013
46. Unified EFI, Inc.: Unified Extensible Firmware Interface Specification, Version 2.6 (2006)
47. VMware: VMware Workstation Pro (2018). https://www.vmware.com/il/products/workstation-pro.html
48. Walters, J.P., et al.: GPU passthrough performance: a comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications. In: 2014 IEEE 7th International Conference on Cloud Computing, pp. 636–643. IEEE. Chicago, June 2014
49. Ye, K., Che, J., Jiang, X., Chen, J., Li, X.: VTestkit: a performance benchmarking framework for virtualization environments. In: 2010 Fifth Annual ChinaGrid Conference, pp. 130–136. IEEE, July 2010
50. Zaidenberg, N.J.: Hardware rooted security in industry 4.0 systems. In: Dimitrov, K. (ed.) Cyber Defence in Industry 4.0 and Related Logistic and IT Infrastructures, Chap. 10, pp. 135–151. IOS Press (2018)
51. Zaidenberg, N.J., Khen, E.: Detecting kernel vulnerabilities during the development phase. In: 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, pp. 224–230. IEEE, November 2015
52. Zhang, L., Wang, L., Zhang, R., Zhang, S., Zhou, Y.: Live memory acquisition through FireWire. In: Lai, X., Gu, D., Jin, B., Wang, Y., Li, H. (eds.) e-Forensics 2010. LNICST, vol. 56, pp. 159–167. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23602-0_14

PVIII

HYPERVISOR MEMORY ACQUISITION FOR ARM

by

Raz Ben Yehuda, Erez Shlingbaum, Shaked Tayouri, Yuval Gershfeld and Nezer Zaidenberg 2021

Forensic Science International: Digital Investigation. 301106

ScienceDirect
www.sciencedirect.com

Digital Investigation 00 (2021) 1–8

RIAI
www.elsevier.es/RIAI

Hypervisor Memory acquisition for ARM

Raz Ben Yehuda[a], Erez Shlingbaum[b], Yuval Gershfeld[b], Shaked Tayouri[b], Nezer Jacob Zaidenberg[b]

[a] *University of Jyväskylä, Jyväskylä, Finland*
[b] *College of Management Academic Studies, Street, Rishon LeZion, Israel*

abstract

Cyber forensics use memory acquisition in advanced forensics and malware analysis. We propose a hypervisor based memory acquisition tool. Our implementation extends the volatility memory forensics framework by reducing the processor's consumption, solves the in-coherency problem in the memory snapshots and mitigates the pressure of the acquisition on the network and the disk. We provide benchmarks and evaluation.

Keywords
Security, Virtualization, Forensics, ARM, Android

1. Introduction

A rootkit is a malware that hides itself along with the malicious payload that it carries(Blunden). Rootkit research is a cat a mouse game in all computing platforms.

Researchers develop better forensics to detect rootkits while others develop state-of-the-art rootkits. Our research improves the way LiME acquires memory images. We enhanced LiME's (lime) memory acquisition tool for Volatility (Farmer) memory forensics software. Our system takes a precise, atomic memory image of an online system without stopping the online system. Furthermore, it is less suspect to acquisition errors as memory acquisition is done outside the operating system. Therefore the acquisition process is not vulnerable to stealth malware hiding.

We describe our design and implementation of an online memory forensics system. We also perform a performance analysis of memory acquisition performance figures on the online system.

2. Background

2.1. Rootkits and stealth malware

Two of the most famous rootkits of recent years were Stuxnet's (Langner) and Sony BMG (Mulligan) rootkit. Stuxnet's purpose was to attack Iran's nuclear enrichment program. Countries and elite intelligence agencies developed Stuxnet for espionage and sabotage purposes. Sony BMG rootkit was designed to install and hide DRM software on end-user machines (Hoglund; rootkit). These two examples demonstrate the rootkits that are no longer "hacker tools", but rather tools employed by countries and top industrial companies for national and business purposes.

Despite having completely different authors and purposes, both software contained a similar concept of masking its existence by hiding the malware files and the running processes. Live memory acquisition is a tool used by forensics researchers to reverse engineer the malware. Forensics researchers attempt to identify the authors, their goals, and any weakness in the malware itself. Volatility (Farmer; Andrew) is an open-source (GPLv2) framework for analysing memory (Aljaedi). It is a forensics toolkit used to analyze memory snapshots. Thus Volatility is often used to detect such hidden malware. (Oktavianto)

2.2. Forensics and Volatility

Volatility is state of the art RAM snapshot analysis software. It is available for Windows, Linux, Mac, and Android, 32bit and partially 64bit. Volatility is based on Python (Sanner), which makes development in Volatility easy for most analysts. Also, Python is available on many operating systems, including Android phones. Volatility availability on Android allows for local memory capture and analysis, saving the need to transmit gigabytes of a RAM snapshot over the network. Volatility API is extensible, and forensics researchers can add plugins easily. Volatility's developers use a reverse engineering approach to understand the acquired memory. Thus, Volatility provides capabilities and information that are not usually possible through standard tools. For example, examining undocumented data structures in windows OS. Volatility supports various formats: crash dumps, hibernation files, VMware's vmem, VMware's saved state and suspended files (.vmss/.vmsn), VirtualBox core dumps, LiME (Linux Memory Extractor), expert

witness (EWF), and direct physical memory over Firewire. Volatility is considered fast compared to other forensics tools. It analyses the entire memory image file in a few seconds.

2.3. LiME

LiME (Linux Memory Extractor) is a Linux kernel module that performs acquisition of volatile memory on Linux distributions and Linux kernel-based devices (Dave), such as Android. Since LiME is a Linux kernel module, it does not require any user-space tools to perform memory captures. Therefore, LiME memory captures are considered sounder than other memory acquisition tools. Also, since LiME is a pure kernel module, it is easy to use it in Android devices and embedded Linux devices in general.

2.4. ARM permission model

ARM has a unique approach to security and privilege levels (Penneman) that is crucial to the implementation of our microvisor. In ARMv7, ARMv8 introduced the concept of secured and non-secured worlds through the implementation of TrustZone (Winter). ARMv8 architecture includes four exception (permission) levels as follows.

Exception Level 0 (EL0) Refers to the user-space. Exception Level 0 is analogous to ring 3 on the x86 platform.

Exception Level 1 (EL1) Refers to the operating system. Exception Level 1 is analogous to ring 0 on the x86 platform.

Exception Level 2 (EL2) Refers to the hypervisor (HYP mode). Exception Level 2 is analogous to ring -1 or real mode on the x86 platform.

Exception Level 3 (EL3) Refers to the TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no direct analogous modes, but related concepts in x86 are Intel's ME or SMM.

Each exception level provides its own set of special-purpose registers and can access the registers of the lower levels, but the not higher levels. The general-purpose registers are shared. Thus, moving to a different exception level on the ARM architecture does not require the expensive context switch that is associated with the x86 architecture.
In the context of the paper, we use EL2 hypervisor to extend LiME.

Our hypervisor's sole purpose is to provide services to the operating system. It does not support running multiple operating systems. It is a thin hypervisor and does not emulate new hardware components. Therefore, the hypervisor has minimal overhead, which we demonstrate later in this paper. Such systems are often called microvisors. Henceforth, We use the term memory acquisition microvisor to describe our solution.

3. Contribution

3.1. High level design

We use Volatility (Rotvold) and LiME to analyse the memory to detect an irregular state. Our contribution focuses on reducing CPU consumption and heat when using LiME. Our memory acquisition microvisor also provides consistent memory images. Also, we ported parts of Volatility to support 64bit ARM Linux kernels.

3.2. Volatility

Volatility currently supports ARMv7 systems. One key difference between ARMv7 and ARMv8 regarding memory acquisition is that ARMv8 usually runs 64bit kernels while ARMv7 runs 32bit. We ported Volatility to ARM64 bit kernels and contributed our modifications to the Volatility community. This approach is important because most phones and Android devices today use ARMv8-a processors and a 64bit Linux kernel.

3.3. Microvised LiME

Our contribution concentrates on LiME. LiME creates a reflection of the computer's RAM by accessing each physical page and writing it to the disk or the network (Algorithm 1). LiME scans the RAM sequentially and allocates an auxiliary page for each page, and copies the page's contents to it. The auxiliary page is transmitted (or written to disk) and then released.

Algorithm 1: LiME Main Transmission

```
while not end of RAM do
    Map the current physical page the to kernel
      memory
    Allocate an auxiliary page
    Copy Physical Page to the auxiliary page
    Transmit the auxiliary Page
    Free auxiliary page
    Unmap the Physical Page
end
```

This paper contribution focuses on solving three disadvantages of LiME:

- The snapshot image in-coherency.
- The processor's high utilisation while the acquisition takes place.
- LiME is unable to predetermine the chances of an incoherent memory snapshot.

3.4. Memory In-coherency

The main memory is likely to change during the acquisition. For example, Figure 1 demonstrates a browser launch during LiME memory acquisition, and a few seconds after the browser immediately exits. The operation changes the operating system's processes table after LiME already transmitted it. Therefore, starting and stopping the browser creates memory in-coherency. Memory in-coherence is a result of processes that exist in memory but not on the process table, pages that belong to a process that does not exist in the table, etc.

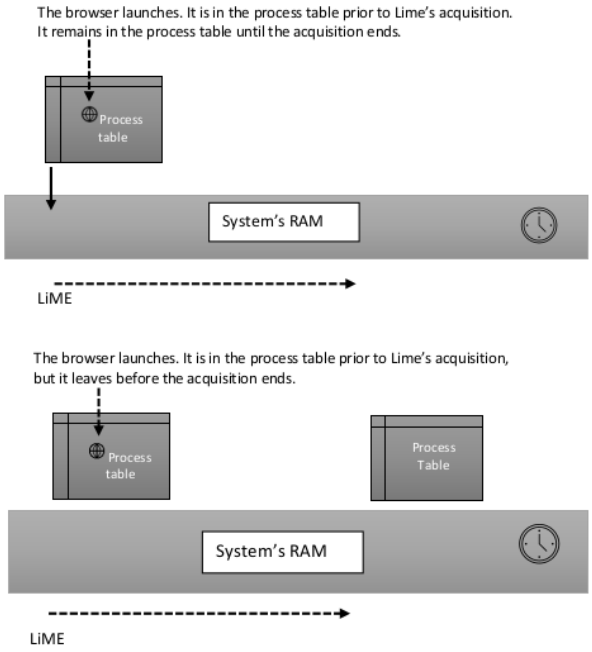


Figure 1: LiME transmission

Andrew et al. (Andrew) describe other in-coherency reasons, such as multiples cores, the increasing amount of RAM and the kernel heuristics, and point the increasing amount of RAM as the reason for page smearing (page smearing is type of memory in-coherence. page smear occur when a page content is changed while it is acquired). In this paper context, (Andrew) mention two techniques to deal with page smearing:

- **Leveraging virtual machine hardware extensions.** Use various virtualization techniques, such as blue pill (a hypervisor that traps an instance of a guest OS without being detected)(Rutkowska; algawi), to acquire the memory.
- **Smear-aware acquisition tools.** Provide the acquisition tool awareness to changes in page tables.

Our technology utilizes virtualization to acquire memory, but without freezing the operating system (hibernation).

To emphasise versatility of the RAM (and page smearing) while LiME transmits, we recorded the first 500 page faults positions while LiME works. Table 1 presents the distance (in pages) between two consequent page faults. The total number of pages is 242000. We performed the test above while the system was idle. We measured for each page fault the distance between the memory addresses of two consecutive page faults.

Avg	Min	Max	Std dev
426	-236990	236921	44837

Table 1: Page faults distribution measured in page offsets

The pages positions distribution is vast. Therefore, we expected to find inconsistencies in the memory image snapshot.

Thus, this paper offers a technique to solve the inconsistency. We wrap the GPOS (General Purpose Operating System, such as Linux or Android) by a minimal virtual machine (VM), and during the acquisition, we record and copy the faulting page content before it is transmitted. In virtual machines, when a guest accesses a page, the Memory Management Unit (MMU) traps it to a secondary page table managed by the hosting machine. This mechanism is referred to as a stage 2 fault. In ARM, the second (stage 2) memory table is called the Intermediate Physical Addresses (IPA) (Penneman).

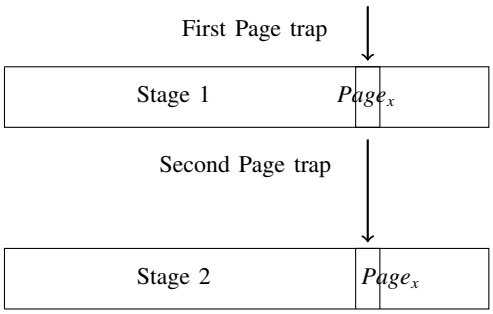


Figure 2: Virtual Machine dual stage memory trap

Thus, to achieve coherency, we set all stage 2 tables to read-only before the memory acquisition starts. This way any write access to any page is trapped to the microvisor. We then execute LiME to acquire the computer's RAM; at this point, any page that traps to the microvisor is copied to a **pages pool** and the real page permissions are set to read-write. Therefore each page can trap to the microvisor at most once.

In addition to our microvisor, our solution includes a modification to the LiME driver. LiME, using our technique, linearly scans the memory, and in each cycle it checks the pages pool to verify that the current page did not already fault. If the page is resident in the pool, LiME transmits the copied page and releases it back to the microvisor's pool for re-use. Algorithm 2 describes LiME's new implementation while using our microvisor. We refer in this paper to this technique as a **linear acquisition**.

3.5. CPU consumption

The linear acquisition has a flaw. As we show later in the evaluation section, the performance cost of linear memory acquisition is very high. To reduce the processor consumption, we modified LiME's main transmission routine to be less CPU intense. In this approach, LiME does not scan the memory linearly. Instead, LiME removes pages from the pool as long as there are pages in it. If there are no pages, it linearly sends pages as before, but it sleeps for a few milliseconds in each transmission cycle. After each cycle, the pool is checked for accessed pages. Any page accessed by LiME is verified that it is not re-transmitted. This is done by checking its permissions bits in the stage 2 table (Figure 2); If the page was transmitted then it is writeable. We refer to this solution as a **non-linear acquisition**.

Algorithm 2: Linear Acquisition: Transmission with a microvisor

```
while not end of RAM do
    Allocate an auxiliary page
    if the physical page has an old version in the pool
     then
        unlink (remove pointer to) the page from pages
          pool
        Copy this page to the auxiliary page
        Place back the page to the pool
        else
            Map Physical Page
            Copy physical page to the auxiliary page
            Unmap Physical Page
        end
    end
    Transmit auxiliary Page
    Free auxiliary page
end
```

3.6. Refraining from incoherent images

As we've shown, LiME produces in-coherent memory dumps. Our microvised LiME solution offers a technique that produces cohesive memory images; However, in cases where there are too many page faults, it is not possible to create a coherent memory image, because the pages pool is overloaded. In this case, we disable the acquisition and start again later. Other techniques to tackle this problem (for example, a dynamic pool) are complex.

3.7. Microvisor memory model

The microvisor can be used to create TEE as we have shown in (benyehuda). Here we explain the ARMv8-a hypervisor memory model in the context of this paper. In ARMv8-a, the hypervisor (EL2) memory table is not accessible to the other exception levels. This means that it is not possible for code running in EL2 to access memory in another exception (Figure 3) level without premature mapping.

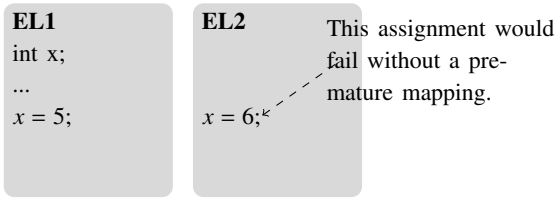


Figure 3: Access from EL2 to EL1

For instance, to access the page that contains the variable x in (Figure 3), from EL2, EL1's pages must be pre-mapped to EL2. However, during the acquisition, each page accessed in EL1 or in EL0 traps to EL2, and therefore, must be copied and its address recorded, to a page in the microvisor pages pool. So it is essential to map this page to the microvisor. We used KVM for ARM (Dall) for this purpose.

Our hypervisor page trap needs to access both virtual userspace and virtual kernel space addresses. Unfortunately, ARMv8-a hypervisor MMU is capable of handling only user-space addresses (the upper 16 bits are zeroes) and does not support kernel virtual addresses (upper 16 bits are ones) at all. So instead of handling virtual addresses, we decided to handle physical addresses. We rely on the following ARM property: When a page traps to EL2, in addition to the virtual address of the faulting page, the physical address of the page is also reported in a special register.

Therefore, we map the entire physical memory to the hypervisor (Figure 4) as-is, meaning, we map each address without any offset. This way, the code, and some data are mapped in a certain part of the address space, represented by a very high number; while the entire computer's RAM resides in address starting from address 0 (the lowest physical address). To summarise, we have two distinct mappings techniques:

- General Mappings

  hypervisor map(kernel addr) = address + offset

  These mappings are used to execute code and access general management data from both the kernel and the hypervisor.

- Physical mappings

  hypervisor map2(physical address) = physical address

  These mappings are used only in the page trap function of the microvisor. The trap copies the data to be used later by LiME.

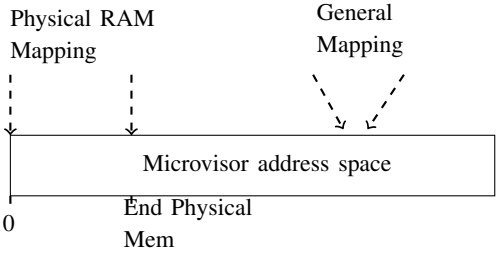


Figure 4: Microvisded LiME's Memory layout

All the pages in the general mapping are mapped in the physical mapping as well. We note that some of these pages are owned by the microvisor. Therefore, they are never accessed by EL1/EL0 and don't trap to EL2 (Our microvisor traps from EL1 and EL0).

3.8. Acquisition Resources Consideration

This section discusses the memory resources required for the acquisition. Figure 5 demonstrates the fluctuations in the number of pages accessed by each processor in our hardware. It is evident that there is a burst in the number of pages as LiME starts and when the acquisition ends.

In our hardware, nearly 600 pages are accessed as the acquisition starts. So, we expect our pool to be in this magnitude. Let

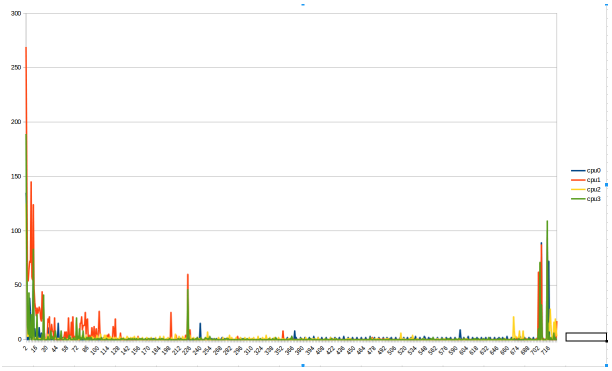


Figure 5: #pages accessed to the microvisor. X axis is #samples

us examine the overhead of 1000 pages assigned to the pool in the hardware of 242000 pages (PI3 - our evaluation hardware, has 940 MB of RAM). Thus the pages pool requires less than half per cent of the total RAM:

$$\frac{1000}{242000} \sim 0.4\%$$

To summarise this section, our acquisition algorithm incurs a temporary negligible overhead of RAM.

4. Evaluation

Our evaluation measures the IPA overhead, offers a performance comparison between the current implementation of LiME to the microvised implementation and demonstrates how RAM in-coherency is solved by our technology.

In the following evaluation, we repeat each test 10 times. We usually provide results in idle mode and then busy mode. Idle mode means that we did not apply any artificial stress on the operating system. A busy mode is when we applied the stress tool *stress* (stress), while LiME was executing, as follows:

```
stress --vm-bytes 128M --timeout 80s --vm 4
```

To measure memory speed, we used RAMspeed (RAMspeed).

4.1. IPA

In Table 2 we attempted to simulate more closely real-world computing load through the use of RAMspeed. A, B and C are locations in the memory. In SCALE m is a constant.

Test	COPY A=B	SCALE A = $m \cdot B$	ADD A + B = C
Single Stage	2.76	1.52	2.60
Dual Stage	2.74	1.52	2.53

Table 2: Real world load GB/s

There is no difference when using a two-stage translation to a single-stage translation. We have shown that IPA does not influence memory access performance.

4.2. Image in-coherency

We first demonstrate the in-coherency of RAM dumps when using LiME without a microvisor. We also show that the microvised LiME provides coherent RAM dumps. We perform the following test. We acquire memory while executing the ”sleep 1” command 50 times sequentially. A consistent snapshot should have a single instance of the sleep process in the process table. Figure 7 and Figure 6 are snippets of Volatility memory analysis of the processes tables. Figure 6 presents the processes tables when the non-microvised LiME is used, while Figure 7 is when using the microvised LiME. In Figure 6 we can see 27 instances of the ”sleep” process while in Figure 7 there is a single instance of ”sleep”. This means that LiME recorded the memory while the process table was changing. In Figure 6 we can see that some of the ”sleep” process ids are successive, which means that no other processes were launched in at least 1 second. This strengthens the claim that even in an idle system, in-coherency is possible.

```
Offset   Name       Pid PPid
------------------ ------------
0x2fc58000 sleep    641  -
0x308f8000 sleep    600  -
0x308f9d00 sleep    601  -
0x308fba00 sleep    602  -
0x308fd700 sleep    603  -
0x30918000 insmod   700  -
0x30919d00 sudo     708  -
0x30948000 swapoff  738  -
0x30978000 sleep    637  -
0x30979d00 sleep    638  -
0x3097ba00 sleep    639  -
0x3097d700 sleep    640  -
0x30980000 sleep    592  -
0x30981d00 sleep    593  -
0x30983a00 sleep    594  -
0x30985700 sleep    595  -
0x309c8000 sleep    596  -
0x309c9d00 sleep    597  -
0x309cba00 sleep    598  -
0x309cd700 sleep    599  -
0x30a10000 sleep    604  -
0x30a11d00 sleep    605  -
0x30a13a00 sleep    606  -
0x30a15700 sleep    607  -
0x30a38000 wpasupplicant 759  -
0x30a39d00 wireless-tools 753 -
0x30ac0000 swapon   735  -
0x30ac1d00 grep     736  -
0x30ac3a00 cut      737  -
0x30b98000 start-stop-daem 731 -
0x30ba8000 sleep    629  -
0x30ba9d00 sleep    634  -
0x30baba00 sleep    635  -
0x30bad700 sleep    636 -
....
```

Figure 6: Processes list reported by Volatility after using non-microvised LiME

```
Offset   Name       Pid     PPid
------------------ -----    -------
0x2fc58000 sleep    641      -
0x3000fa00          318      -
0x3001e700 btuart   319      -
0x30035700 modprobe 508      -
0x30051000 ksoftirqd/0  9    -
x30053700 ksoftirqd/1  17    -
0x3007f000 ksoftirqd/2  22   -
0x304e0000 bash 527  -  0    -
0x304e1d00 top  530 -   0    -
..
```

Figure 7: Processes list reported by Volatility after using Microvised LiME

Now, we want to have some measures for the possible in-coherency while LiME executes. So, we examine the number of EL2 page faults (Table 3) in busy mode and idle mode.

	Avg	Max	Min	Std dev
Idle Mode	130	129	132	1.17
Busy mode	128212	135891	122771	3493

Table 3: Total number of page faults during LiME

Table 3 was produced by the average and other statistical measures of 10 runs. We note that for a pool of 1000 pages the in-coherency should be less than 0.4%.
In idle mode:

$$\frac{130}{242000} \sim 0.00005$$

but in busy mode:

$$\frac{128212}{242000} \sim 0.52$$

The amount of in-coherency in the extreme case is approximately 50%. Therefore, when assessing our model (pool of 1000 pages) to the system's load, the load can be $1000/130 \sim 7.5$ busier than the idle mode presented here, and $1000/128212 \sim 0.0008$ times smaller than busy mode. Evidently, it is more efficient to run LiME when the system is idle.

4.3. Linear acquisition

We compare the performance of linear acquisition of microvisied LiME to the non-microvised LiME. We measure the duration in seconds, in idle mode and busy mode, and the processor consumption in idle mode. In the tests presented in table 4 we used linear acquisition.

LiME mode	Avg	Min	Max	Std dev
Without a microvisor				
Idle mode	74	72	76	1.4
Busy mode	84	83	85	0.7
With a microvisor				
Idle mode	80	74	96	7.68
Busy mode	83	81	85	0.9

Table 4: Duration of the memory dump, idle/busy modes

From table 4 it is evident there is an overhead of 6% when using the microvised LiME. The reason is that for each page faulting to the microvisor, we scan the pool.
Lastly, we measure processor utilisation with and without a microvisor.

LiME mode	Avg	Min	Max	Std dev
no microvisor	52	50	53	1
With a microvisor	48.8	39	52	4.5

Table 5: Processor utilisation in Idle mode

Table 5 presents the overhead of the processor utilisation when LiME acquires memory. There is little difference between, so it is evident that even the linear version of microvised LiME does not incur any significant performance risks.

4.4. Non-Linear acquisition

Here we demonstrate the processor's consumption and the time it takes to perform a nonlinear acquisition. The variable we change in table 6 and is the delay duration in each cycle. We provide measures of the processor's consumption and the duration of the entire acquisition.

Test Conf	Duration in seconds	% Processor Consumption
10 ms	4860	17
20 ms	6600	12
50 ms	14520	5

Table 6: Duration of non linear acquisition

Table 6 proves that it is possible to perform an acquisition with very little processor consumption. The trade-off is a considerably long acquisition duration, in some cases over an hour. An additional benefit of this technique is that it has a low I/O load. The method does not congest the disk, or in the network case, mild network traffic. Thus it is possible to run the acquisition over a cellular medium.

5. Related work

In recent years it was proposed to use hypervisors for many security purposes such as machine introspection and debugging (Zaidenberg). The introspection of virtual machines by the hypervisor was researched heavily. Libvmi (Payne) is a library that provides such introspection services under KVM. It provides VM based snapshots and has an integrated volatility plugin. It was also suggested to use Lguest(Russel) or Xen(Barham) for detection of kernel bugs(Khen), profiling(Menon), Hypertracing (Benbachir), security issues (Zaidenberg)), and access the guest's memory through a thin hypervisor for remote attestation as suggested by Kiperberg et al. (Kiperberg). Forenvisor (Qi) uses the hypervisor to grab and store forensics data on the cloud for later inspection. Kiperberg et al. (Kiperberg) provided a system for atomic memory acquisition and guaranteed atomic access in x86 architecture. Kiperberg et al. approach aimed to solve to main problems, multiprocessing, and ASLR (address space layout randomization). To handle multiprocessing in x86, Kiperberg et al. used a special API to synchronize access to the page table on all processors. Our solution handles this differently, the page table is shared on all processors, and access to the page table is synchronized using spinlocks. Regarding ASLR, this paper assumes that ASLR is disabled on the suggested system.

Andrew et al. (Andrew) discuss page swapping and demand paging as another obstacle to complete the acquisition of memory. Our technique does not acquire swap space or fetches pages of incomplete processes. At this stage, it is not clear whether the Volatility framework is capable to incorporate swap space and on-demand pages.

In Microsoft Windows operating systems, projects, like Powershell Empire (powershell), exploit Windows PowerShell. Powershell Empire does not execute the PowerShell executable file from the file-system but runs directly from the memory. Empire provides keylogging, credential theft, and more. As a result, memory forensics tools fail to detect the execution of PowerShell and forced to rely on pattern matching and search for side-effects of PowerShell post-execution. Another challenge is the Windows .NET framework. .NET runtime is embedded in the process's address space and, therefore, can execute an injected malware (Santiago; Andrew). At the moment there is no available technology that can detect this malware. Additionally, .NET supports function overriding; a technique that malware can manipulate by replacing callbacks with malicious functions.

In Android, a large effort is in the analysis of the Dalvik engine. Researchers created forensics tools (Macht) for Dalvik. Unfortunately, Dalvik was replaced by ART (Android Runtime). Unlike Dalvik which is based on JIT (Just in Time compilation), ART (art) produces ELF binaries. To the day of writing, there is no published forensics technology for ART.

There have been multiple suggestions for memory inspection and acquisition through dedicated firmware. (Zhang) et al. describe such memory acquisition through RDMA. In this paper, we assume that such hardware is not available.

6. Conclusions

We have shown that it is possible to perform a memory acquisition without saturating the processor. Despite the intense processor usage, our memory dumps remain coherent. We also showed that the acquisition is possible without overwhelming the disk or the network, and therefore it is possible to perform it over wireless embedded devices, mainly mobile phones. Our code footprint is considerably low, about 2200 lines of microvisor code.

References

[Penneman] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *Journal of Systems Architecture*, 59(3):144–154, 2013.

[Flur] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, P. Sewell, Modelling the armv8 architecture, operationally: concurrency and isa, in: ACM SIGPLAN Notices, volume 51, ACM, pp. 608–621. 2016

[Blunden] B. Blunden, The Rootkit arsenal: Escape and evasion in the dark corners of the system, Jones & Bartlett Publishers, 2012.

[Langner] R. Langner, Stuxnet: Dissecting a cyberwarfare weapon, IEEE Security & Privacy 9 (2011) 49–51.

[Mulligan] D. K. Mulligan, A. K. Perzanowski, The magnificence of the disaster: Reconstructing the sony bmg rootkit incident, Berkeley Tech. LJ 22 (2007) 1157.

[Hoglund] G. Hoglund, J. Butler, Rootkits: subverting the Windows kernel, Addison-Wesley Professional, 2006.

[rootkit] Davis, Michael and Bodmer, Sean and LeMasters, Aaron, Hacking exposed malware and rootkits, 2009 McGraw-Hill, Inc.

[Farmer] D. Farmer, W. Venema, Forensic discovery, Addison-Wesley Professional, 2009.

[Sanner] M. F. Sanner, et al., Python: a programming language for software integration and development, J Mol Graph Model 17 (1999) 57–61.

[Dave] R. Dave, N. R. Mistry, M. Dahiya, Volatile memory based forensic artifacts & analysis, Int J Res Appl Sci Eng Technol 2 (2014) 120–124.

[Winter] J. Winter, Trusted computing building locks for embedded linux-based arm trustzone platforms, in: Proceedings of the rd ACM workshop on Scalable trusted computing, ACM, pp. 21–30. 2008

[Rotvold] E. D. Rotvold, D. R. Lattimer, M. J. Green, R. J. Karschnia, M. A. Peluso, Interface module for use with a modbus device network and a fieldbus device network, 2007. US Patent 7,246,193.

[lime] J. Sylve, Android mind reading: Memory acquisition and analysis with dmd and volatility, in: Shmoocon 2012.

[Penneman] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, K. De Bosschere, Formal virtualization requirements for the arm architecture, Journal of Systems Architecture 59 (2013) 144–154.

[Dall] C. Dall, J. Nieh, Kvm/arm: The design and implementation of the linux arm hypervisor, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, ACM, New York, NY, USA, 2014, pp. 333–348.

[Dall] C. Dall, J. Nieh, Kvm/arm: the design and implementation of the linux arm hypervisor, in: ACM SIGARCH Computer Architecture News, volume 42, ACM, pp. 333–348. 2014

[stress] Stress-ng, 2017, King, Colin Ian http://kernel.ubuntu.com/git/cking/stressng. git/(visited on 28/03/2018)

[RAMspeed] RAMspeed, a cache and memory benchmarking tool, 2011, Hollander, Rhett M and Bolotoff, Paul V

[Zhang] L. Zhang, L. Wang, R. Zhang, S. Zhang, Y. Zhou, Live memory acquisition through firewire, in: International Conference on Forensics in Telecommunications, Information, and Multimedia, Springer, pp. 159–167. 2010

[Zaidenberg] N. J. ZAIDENBERG, Hardware rooted security in industry 4.0 systems, Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures 51 (2018) 135.

[Payne] B. D. Payne, Simplifying virtual machine introspection using libvmi, Sandia report (2012) 43–44.

[Russel] R. Russel, lguest: Implementing the little linux hypervisor, OLS 7 (2007) 173–178.

[Barham] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: ACM SIGOPS operating systems review, volume 37, ACM, pp. 164–177. 2003

[Khen] E. Khen, N. J. Zaidenberg, A. Averbuch, E. Fraimovitch, Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation, in: 2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), IEEE, pp. 78–85.

[Menon] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, W. Zwaenepoel, Diagnosing performance overheads in the xen virtual machine environment, in: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, ACM, pp. 13–23. 2005

[Benbachir] A. Benbachir, M. R. Dagenais, Hypertracing: Tracing through virtualization layers, IEEE Transactions on Cloud Computing (2018).

[Zaidenberg)] N. J. Zaidenberg, E. Khen, Detecting kernel vulnerabilities during the development phase, in: 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, IEEE, pp. 224–230.

[Kiperberg] M. Kiperberg, R. Leon, A. Resh, A. Algawi, N. Zaidenberg, Hypervisor-assisted atomic memory acquisition in modern systems, in: International Conference on Information Systems Security and Privacy, SCITEPRESS Science And Technology Publications. 2019

[Qi] Z. Qi, C. Xiang, R. Ma, J. Li, H. Guan, D. S. Wei, Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics, IEEE Transactions on Cloud Computing 5 (2016) 443–456.

[Aljaedi] A. Aljaedi, D. Lindskog, P. Zavarsky, R. Ruhl, F. Almari, Comparative analysis of volatile memory forensics: live response vs. memory imaging, in: 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing, IEEE, pp. 1253–1258.

[Oktavianto] D. Oktavianto, I. Muhardianto, Cuckoo malware analysis, Packt Publishing Ltd, 2013.

[art] Android, 2016. art and dalvik., 2020.

[powershell] Bpowershell empire. (n.d.)., 2020.

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

[Wueest] Candid Wueest and Doherty Stephen. The increased use of powershell in attacks. *CA: Symantec Corporation World Headquarters*, pages 1–18, 2016.

[Santiago] Santiago M Pontiroli and F Roberto Martinez. The tao of .net and powershell malware analysis. In *Virus Bulletin Conference*, 2015.

[Macht] H Macht. Dalvikvm support for volatility, 2012.

[algawi] Creating Modern Blue Pills and Red Pills. Algawi Asaf ,Kiperberg Michael, Leon, Roee, Resh Amit, Zaidenberg Nezer, ECCWS 2019 18th European Conference on Cyber Warfare and Security, 6, 2019, Academic Conferences and publishing limited

[lime] Android Mind Reading: Memory Acquisition and Analysis with DMD and Volatility, Sylve, Joe, Shmoocon 2012, 2012

[Rutkowska] Introducing blue pill, Rutkowska, Joanna, The official blog of the invisiblethings. org, 22, 23, 2006

[benyehuda] , Protection against reverse engineering in ARM, Yehuda, Raz Ben and Zaidenberg, Nezer Jacob, International Journal of Information Security, 19 1, 39–51, 2020, Springer

PIX

OFFLINE NANOVISOR

by

Raz Ben Yehuda and Nezer Zaidenberg

submitted

THE OFFLINE NANOVISOR

Abstract: Current real-time technologies for Linux require partitioning for running RTOS alongside Linux or an extensive kernel patching. The Offline Nanovisor is a minimal real-time library OS in a lightweight hypervisor under Linux. We describe a Nanovisor that executes in an offline processor. An offline processor is a processor core that is removed from the running operating system. The offline processor executes userspace code through the use of a hyplet. The hyplet is a Nanovisor that allows the kernel to execute userspace programs without delays. The combination of these two technologies offers a way to achieve hard real-time in standard Linux. We demonstrate high-speed access in various use cases using a userspace timer in frequencies up to 20 kHz, with a jitter of a few hundred nanoseconds. We performed this on a relatively slow ARM processor.

Keywords: Hypervisor; real-time; ARM; Virtualization; Embedded Linux.

Biographical notes:

1 Introduction

Obtaining predictable latency while processing data is a challenging task. That is especially true in general-purpose operating systems (GPOS). In GPOS the program becomes less predictable, in many cases, due to cache and TLB misses Bennett and Audsley [2001]. Software architects handle this unpredictability with various techniques. Such techniques include microkernels, microvisors and partitioning or real-time extensions to the GPOS. Other solutions include auxiliary processors, such as DSPs and even GPUs.

This paper offers an alternative approach to the above for multi-processor machines. By virtually removing a processor from the operating system and running a single program in it, it is possible to achieve predictable latency in a GPOS. Removing a process from the GPOS scheduler and assigning it to a single task is referred by Ben-Yehuda and Wiseman [2013] as the 'offline scheduler'. A difficulty with the offline scheduler design is that it can only run kernel code - in what we refer to as an 'offlet'.

We evolve the offline scheduler to process hyplets as introduced in Ben Yehuda and Zaidenberg [2018]; Ben Yehuda et al [2020]. Instead of running in kernel mode, the offlet executes in hypervisor mode. Hyplet technology is an excellent fit for the offline scheduler because both provide complementary real-time advantages. Furthermore, both run without interrupts, so it was easy to combine them.

Microcontrollers trap real-world events. These microcontrollers send interrupts to the GPOS processor to inform about incoming events. Thus, the accuracy of the data also relies on the rate of the interrupts processed by the GPOS processor. In this context, therefore, we demonstrate the offline scheduler as a technique to acquire data from external devices through high-speed data sampling. We construct test cases in which the physics changes so fast that without a tight loop to access the data, it is not possible to observe these changes. For example, we show that we can measure the beginning and the return of an ultrasonic wave more accurately than a standard Debian Linux. Another essential use we demonstrate is a 20 kHz software timer. High-speed timers are widely used in many real-time applications and

usually require dedicated hardware and software. An interrupt triggers the system's timer and, on Linux, the interrupt usually wakes a user-space process; this entire chain of events causes latency. While the hyplet eliminates kernel-to-user latency, the Offline Nanovisor eliminates the **offlet-to-user latency**. This motivates us to escalate privileges.
Therefore, we provide a hard real-time library in a Linux GPOS and a fast access to sensors that cannot generate interrupts, i.e. sensors that need to be polled. In the Offline Nanovisor, the kernel runs with interrupts disabled. Thus, the Offline Nanovisor guarantees latency as long as the program does not wait for another program or overflow the processor's L1 caches. An executing program never leaves the processor, and it is up to the programmer to yield the processor. As we show later, as long as the code and data remain in the processor's cache, real-time responsiveness is guaranteed. Furthermore, if the data size surpasses the processor's cache size, we can pre-fetch it (or part of it). The pre-fetch is possible because there is no other program in the processor. Therefore, it possible to predict which buffers are needed and when.
The outline of this paper is as follows:

- **Section 2** describes hyplet and offlet technologies and their combination.
- **Section 3** presents the Offline Nanovisor and its programming model.
- **Section 4** is an evaluation.
- **Section 5** demonstrates some use cases for the Offline Nanovisor.
- **Section 6** provides an overview of related work.
- **Section 7** presents a summary.

2 Background

This section describes the offline scheduler and the hyplet.

2.1 The offline scheduler

In many Linux devices today, unplugging a processor is a way to reduce power consumption and heat. An unplugged processor is stripped out of any resources it controls, such as memory and scheduling system, and then is moved to sleep mode. In Intel architectures, the unplugged processor runs a loop of the halt instruction. In ARM, the processor switches to EL3, the TrustZone, which relaxes the processor or executes the WFE (Wait For Event) instruction. At this point, the offline scheduler appears. The processor invokes a kernel driver procedure instead of shifting to a relax mode. We depict offline scheduling in Algorithm 1.

Algorithm 1: ***Offline scheduler typical main loop***

```
..Drop the processor procedure
   while Processor in offline mode{
     user_callback()
     pause()
}
```

This kernel driver is the offline scheduler. The offline scheduler cyclically assigns the processor a single task. The offline CPU operates without interrupts and can access the entire kernel address space natively. Thus, a program executing in an offline mode may access most native APIs of the Linux kernel as long as these APIs do not rely on the processor to be online. For example, it is not possible to use the standard kernel memory allocation (kmalloc), but freeing memory is possible. The offline scheduler, depicted in Figure 1, shows that the offline processor, CPU3, can access the entire address space in the operating system like a program in the online processors. The online processors, in the light gray area, run the GPOS while the offline processor, in the darker gray area, does not run the GPOS. The GPOS does not control CPU3 in the gray area. In Linux, it is possible to unplug no more than N-1 processors in an N processors computer.

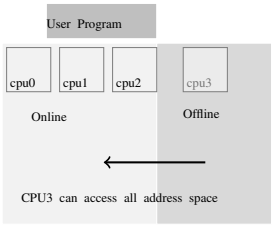


Figure 1 Offline scheduler

Waiting for an event using a tight loop is considered to be a bad technique. There are several reasons for this:

- The processor does not serve any other task, which violates the multiprogram paradigm described by Betz et al. [2009]. A single program cannot keep either the CPU or the I/O devices busy at all times.
- The processor's temperature increases.
- The processor must go through a quiescent state (Bovet and Cesati [2005]). In operating systems, a quiescent state is when the program performs a system call or relinquishes the processor. If the program does not relinquish the processor, then the operating system would get into an error state and might hang.

For these reasons, we decided to hot-unplug the processor because unplugging relaxes some of the above as the processor is not part of the operating system kernel and, therefore, not subordinate to its heuristics. To mitigate the heat problem, we called the *pause* mnemonic instruction in x86 and its ARM equivalent mnemonic, *yield*. The duration of the delay instructions is measured so they can be used accurately.
Historically, the first use of the offline scheduler was as a real-time packet scheduler. We created the offline scheduler as a component of a high-performance video server. Thus we proposed a way to achieve real-time alongside the GPOS. Our method does not have the costs and efforts of an additional RTOS.

Ad hoc RTOS for power saving
Due to their nature, real-time operating systems usually consume more power Shalan and El-Sissy [2009]; Madhavapeddy et al. [2015]. Thus RTOS generates more heat than

a general-purpose operating system. Vendors deal with these problems through the use of auxiliary processors such as DSPs and GPUs and power-saving software Datta et al. [2012]; Paul and Kundu [2010] and hardware. The Offline Nanovisor behaves like these auxiliary processors. the GPOS manages the processor as long as there is no need for real-time performance. In these periods (when real-time performance is not required), the processor is usually unplugged or running in reduced frequency. Once there is a need for real-time, the Offline Nanovisor unplugs the processor and executes a hyplet in it. We refer to this process as 'offlet booting'. Unplugging a processor takes about 100 milliseconds. For returning the CPU to the GPOS, the Offline Nanovisor unmaps the hyplet and then returns the processor to the GPOS. This process also takes approximately 100 milliseconds.

Resource sharing
Resource sharing between two VM guests, a host and a guest VM, or a complete unikernel with the GPOS may be costly in terms of engineering effort (device drivers) and synchronization (shared memory). The Offline Nanovisor, however, constantly switches between kernel mode and HYP mode. To utilize the host's services, the offlet accesses the entire Linux system resources natively and securely. The same applies for the hyplet, it also runs natively but in the user process's confined address space. In both cases, the offlet and hyplet programs access variables directly.

Learning Curve
The offline hyplet learning curve is small. It resembles setting a UNIX signal; it does not require any special compilers or tools, and communication between the Linux process to the real-time context is not needed as they share address space. In contrast, assimilating other RTOS systems in some cases is a big engineering effort that requires high expertise.

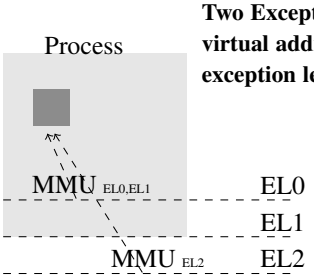
3 The hyplet

ARM8v-a specifications offer to distinct between user-space addresses and kernel space addresses by the MSB (most significant bits). The user-space addresses of Normal World and the hypervisor use the same format of addresses. These unique characteristics are what make the hyplet possible. The Nanovisor can execute user-space position-independent code without preparations. Consider the code snippet at Figure 2. The ARM hypervisor can access this code's relative addresses (adrp), stack (sp_el0) etcetera without pre-processing. From the Nanovisor perspective, Figure 2 is a native code.
Here, for example, address 0x400000 might be used both by the Nanovisor and the user.

```
400610: foo:
400614: stp x16, x30, [sp,#-16]!
400618: adrp x16, 0x41161c
40061c: ldr x0, [sp,#8]
400620: add x16, x16, 0xba8
400624: br x17
400628: ret
```

Figure 2 A simple hyplet

So, if we map part of a Linux process code and data to a Nanovisor it can be executed by it.
When interrupt latency improvement is required, the code is frequently migrated to the kernel, or injected as the eBPF framework suggests Miano et al. [2018]. However, kernel programming requires a high level of programming skills, and eBPF is restrictive. A different approach would be to trigger a user-space event from the interrupt, but this would require an additional context switch. A context switch in some cases is time-consuming. We show later that a context switch is over 10 μs in our evaluation hardware. To make sure that the program code and data are always accessible and resident, it is essential to disable evacuation of the program's translation table and cache from the processor. Therefore, we chose to constantly accommodate (cache) the code and data in the hypervisor translation registers (Figure 3) in EL2 cache and TLB Penneman et al. [2013]. To map the user-space program, we modified the Linux ARM-KVM, Dall and Nieh [2014] mappings infrastructure to map a user-space code with kernel space data.



Two Exception Levels access the same physical frame with the same virtual address of some process. However, the page tables of the two exception levels are not identical.

Figure 3 Asymmetric dual view

Figure 3 demonstrates how identical addresses may be mapped to the same virtual addresses in two separate exception levels. The dark shared section is part of EL2 and therefore accessible from EL2. However, when executing in EL2, EL1 data is not accessible without previous mapping to EL2. Figure 3 presents the leverage of a Linux process from two exception levels to three.

3.1 Programming model

The offline hyplet requires modifications in the native C code. A hyplet'ed program that interacts with hardware devices accesses these devices through the kernel in the offline processor. A common hyplet is one that constantly exits the Nanovisor to the kernel and from the kernel back into the nanovisor. Another possibility is for programmers who are acquainted with eBPF programming and can write an abstraction for the kernel part in eBPF. Thus, porting of a real-time program includes kernel programming and user-space programming. In general, whenever there is need for an operating service, we exit the Nanovisor to the GPOS. Algorithm 2 demonstrates real porting; it is the simplified implementation of the ultrasonic sensor code we use later in this paper. The code accesses two different GPIOs (General Purpose I/O), so it needs to perform some system calls.

Algorithm 2: ***Native C program for an ultrasonic sensor***

```
 // triggers the sound wave
write(fd_trig, "1", 1)
do {
// Wait for the wave transmission
   read(fd_echo, &in, 1)
   t1 = get_time()
} while (in == 0);
do {
// Wait for the wave end of reception
   read(fd_echo, &in, 1);
   t2 = get_time();
} while (in == 0);
// Return the delta to the user program
return t2 - t1
```

To replace system calls, the Nanovisor exits to the kernel and performs the required service in kernel mode. An exit to the kernel is done through the use of $ERET$ and the return back to the Nanovisor is done by HVC.

Algorithm 3: ***Ultrasonic Hyplet***

```
EL2: long hyplet_timer(long ts)
EL2: {  stash the time stamp and wait ..
EL2:     timestamps[i]=ts;
EL2:     wait(N microseconds);
Done waiting. Exit the Nanovisor(EL2). Move to EL1;
EL2: }
...
EL1: long user_callback()
EL1 {
EL1:     gpio_set_value(gpio_trig, val)
EL1:     do {
EL1:       val = gpio_get_value(gpio_echo)
EL1:       t1 = get_time()
EL1:     } while (in == 0)
EL1:     do {
EL1:       in = gpio_get_value(gpio_echo)
EL1:       t2 = get_time()
EL1:     } while(in == 0)
 The callback finished. The framework enters the Nanovisor
    and the arguments are passed to the hyplet.
EL1: }
```

Therefore, the programming model requires that the program must change and be broken down to hyplets and offlets, which are then called sequentially, thereby maintaining the program state. For instance, in Algorithm 3, we perform the I/O in kernel mode in an offlet and then return to the hyplet to process the new data in user mode. Each line of code in Algorithm 3 is prefixed with the exception level the processor is in when it executes it. The programmer does not need to perform $ERET$ or HVC by themselves. The programmer

registers a user callback when they write the kernel portion. When the callback returns, the framework returns to the Nanovisor that runs the next hyplet.

The hyplet framework offers the following APIs:

- Memory mapping
  hyp_map(address,size)
  hyp_map_all()
  hyp_unmap(address)
  Maps or unmaps regions of code or data. It is also possible to map the entire process's address space, but we discourage it because usually the address space grows during the process life.

- Stack assignment
  hyp_set_stack(addr,size)
  A user should map some memory chunk as a stack.

- vma mapping
  hyp_map_vma(addr, size)
  Maps an address only if it is a vma. A vma (virtual memory area) is a virtual contiguous memory Bovet and Cesati [2005] with the same properties (read/write/execute).

- Offlet activation
  hyplet_drop_cpu(cpu_id)
  Unplugs a processor from the kernel. Uses Linux standard hotplugging API, i.e. we utilize Linux sysfs to remove a processor.

- Hyplet assignment
  hyp_assign_offlet(cpu_id, user function)
  hyp_unassign_offlet(user function)
  The user-space provides a function as an hyplet to execute in cpu cpu_id. Once the hyplet is assigned to the offline processor, it would run.

- Synchronisation
  hyp_lock(spinlock),hyp_unlock(spinlock)
  In cases where the programmer wishes to protect a resource from concurrent access from EL0 and EL2 or concurrent access from two offlined processors.

- Get time
  hyp_gettime()
  Returns the current time in nanoseconds. It is the value of the cntvct_el0 register that holds the current clock value.

- Printing
  hyp_print(const char* fmt,...)
  Prints in hyplet context.
  print_hyp()
  Records the print string and the values to a temporary buffer in EL2. When the program is in EL0, it should call print_hyp to print the data to the program's standard output as if it were a regular C's printf(3).

- Event
  hyp_wait()
  Waits for the completion of the offlet. *hyp_wait()* is similar to the UNIX wait(2) system call. There is no restriction on the number of callers.

EL1 data are passed to and from the hyplet through the function's arguments.
When the process exits, the hyplet is automatically removed from the Nanovisor. At this point, the processor remains offline, waiting for a new assignment.

```
while (offlet_assigned == 0) {
    pause();
}
```

In the kernel, the offlet API uses $struct\ hyp\_wait$ that contains the user callback and some additional context information. We provide two APIs for offlets: *offlet_regiser(hyp_wait, cpu_id)* and *offlet_unregister(hyp_wait, cpu_id)*. These two APIs may be used after a processor is unplugged or before.

Delicate mapping
There are times when we want to map only certain global variables and functions to the nanovisor. For this, we used gcc sections. For example:

```
__attribute__ ((section("hyp"))) unsigned int a = 0;
unsigned int b = 0;
```

In this case, we want only to map the variable 'a' and not 'b'. So, we grab the ELF (Executable Linkable Format) section 'hyp' and map it to the Nanovisor. The Offline Nanovisor offers a library that reads the program ELF structure and maps the required sections. For example, the below maps section 'hyp'.

```
Elf_parser_load_memory_map("myprogram")
get_section_addr("hyp", &hyp_sec, &sec_size);
hyplet_map_vma((void *)hyp_sec, cpu_id)
```

3.2 Runaway hyplet

If the hyplet is locked in an endless loop, we need to intervene in the code executing in the processor. For this, we offer to write on the entire EL2 user-mapped code through a different processor an opcode that generates an abort. Once the offline processor executes this code, the processor aborts, records the program counter and exits gracefully from EL2. For the programmer to understand that an infinite loop exists, the hyplet calls a function that increments a counter. This counter is shared between the EL2, EL1 and EL0. The value of this counter and the value of the program counter are visible through the Linux procfs file system. So, if this counter is not updated, the programmer can quickly locate the infinite loop and its position in the program.

3.3 Soft microcontroller

The Offline Nanovisor may be described as a software microcontroller. Instead of having the operating system kernel serve an interrupt from an external microprocessor, the hyplet

is the one that decides whether or not to interrupt the **user process**. In the phase of research and development, a soft microprocessor can be used during a proof-of-concept phase and before the electronics design, thus reducing costs.
Furthermore, some devices do not have any interrupts at all. Instead of receiving an interrupt, the device driver cyclically accesses the device. However, there is no way to know when data are available except by regularly accessing it in a tight loop. Cyclic pulling of the device data implies jitter when performed by the GPOS because we may not write a tight loop un-interrupted as noted previously.

3.4 AMP cache thrashing

AMP, or Asymmetric Multiprocessing Architecture, is when the system cores run different operating systems (a notable example is the Jailhouse microvisor Blackham et al. [2011]). AMP cache thrashing occurs when a GPA (guest physical address) for two different guests is mapped to different host physical addresses. The GPA might map to the same L2 or L3 cache lines due to cache associativity. Because the two guests do not have the same data in the GPA, they disturb each other, which degrades performance. This problem does not happen in the Offline Nanovisor because the hyplets use EL2 cache lines and not EL1's, and the offline processor is not virtualised.

3.5 Concurrency

Linux supports unplugging multiple processors. Thus, it is possible to load the Offline Nanovisor on multiple processors concurrently. The Offline Nanovisor API provides a synchronisation primitive, a spinlock, for the case where a process is shared between multiple processors' cores.

4 Evaluation

We demonstrate different types of experiments. The first is a software timer in which we show the accuracy of the Offline Nanovisor. In the next two sections, we demonstrate how some systems today do not reflect accurately enough the natural non-discrete physical world. The two experiments show that even when we use microcontrollers connected to a small computer that runs Linux, we lose accuracy due to the nature of the Linux operating system. We conduct our experiment with off-the-shelf devices, many of which are used in real products and can be reproduced.
We conducted measurements on a Raspberry Pi3 running a standard Linux OS. For practical reasons, we compare the Offline Nanovisor only to Linux and to technologies available in ARM. Dune Belay et al. [2012] and Rump Kernels Kantee et al. [2012] are not available on ARM at the time of writing; RTAI (last released in Feb 2018) is not available for PI and not for 64-bit ARM.
Table 1 summarizes the Raspberry Pi3 main specifications:

Soc	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2GHz
RAM	1GB LPDDR2 (700 MHz)
Oscillator	19.2 Mhz

Table 1 Pi3 Specifications

It is also important to note that the Pi's clock inaccuracy is up to 140 ppm.

4.1 Latency

Linux handles interrupts Regnier et al. [2008] in two parts:

top half that acknowledges the interrupt

bottom half that handles the interrupt

So, before we start with the experiments, we first need to understand the variation in the PI3 interrupt's latency before acknowledgement of the interrupt itself, i.e. the top half. As the source of the interrupt, we used an Invensense mpu6050 Fitriani et al. [2017] IMU (inertial measurement unit) to the Pi3, and configured it to work in the i2c protocol. In i2c, for every 8 bits of data, there is an acknowledgement signal that generates an interrupt to the Pi3. The acknowledgement signal is sent on the SDA line (the data line of the i2c) and received by the operating system kernel. We wanted to measure the time interval between the moment of the **i2c ACK** and the moment the processor runs the main interrupt routine. Therefore, we connected a logic analyzer probe to the SDA of the IMU device. We programmed one of the Pi's GPIOs to trigger a signal in the kernel's main interrupt routine. The results were an average of 3.9 μs, a maximum 9μs and a minimum 1.7 μs. Therefore, interrupts reach the service routine at varying times. Our results emphasise the fact that there are occasions when we cannot rely on an interrupt to reach the kernel in a predictable time. In other words, while a jitter of 10 μs in a 1 ms cycle may be considered tolerable in some cases, in a 100 μs cycle it is not.

5 Use cases

Next, we demonstrate several use cases.

5.1 Timer

We first evaluate other real-time operating system technologies for ARM and check their jitter at 1 kHz tick rate. We evaluate seL4 Wang et al. [2015] and Xvisor Patel et al. [2015] because they offer RTOS for ARM, Linux RT_PREEMPT RTOS, a standard Debian Linux and the Offline Nanovisor. Standard Linux is an off-the-shelf Debian Linux without any kernel modifications. The test is a simple 1-millisecond timer program written in C for each operating system.
In Figure 4, other than seL4 and the hyplet (the Offline Nanovisor), none of the other solutions can handle hard real-time. We note that we performed this test on an idle system without any real load, and took time samples from the system clock.

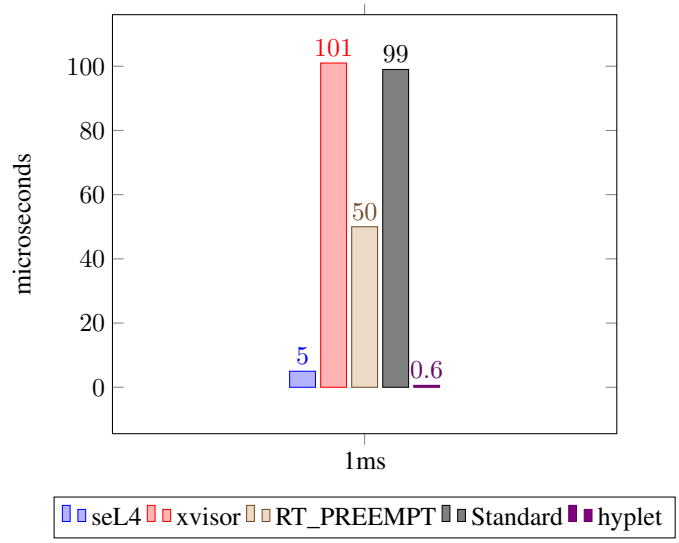


Figure 4 Maximum jitter in μs in idle mode of various operating systems

We did not continue the evaluations in higher frequencies, such as 20 kHz, because at these rates the operating systems are not responsive. We now want to evaluate the Offline Nanovisor. The Offline Nanovisor latency must resemble a real microcontroller latency. For example, Renesas MCU (microcontroller unit) MC16C/62P, has a 24 MHz processor, and according to Anh and Tan [2009], when it uses uTkernel, its latency from interrupt to task is about 1 μs. So, to be a competitive substitute, we aim for a few microseconds interrupt to task latency. As we will soon see, we reach this goal.

The offlet-hyplet test program is a timer function in HYP mode that waits for the time to expire, returns to EL1 to toggle a GPIO in EL1 and then returns to EL2 to wait again.

We measured latencies at various intervals. We constructed each test in two modes: An *idle mode* - While the processors are mostly idle and a *busy mode*. When in *busy mode* we flooded the network interface card (100 MbE) with network traffic to disturb the processors and consume bus cycles. The rate of interrupts was 3000 INTR/second, whereas in idle mode the network generates about 300 INTR/second.

We used an oscilloscope to achieve nanosecond accuracy. Figures 5 and 6 present the jitter in sub-microseconds.

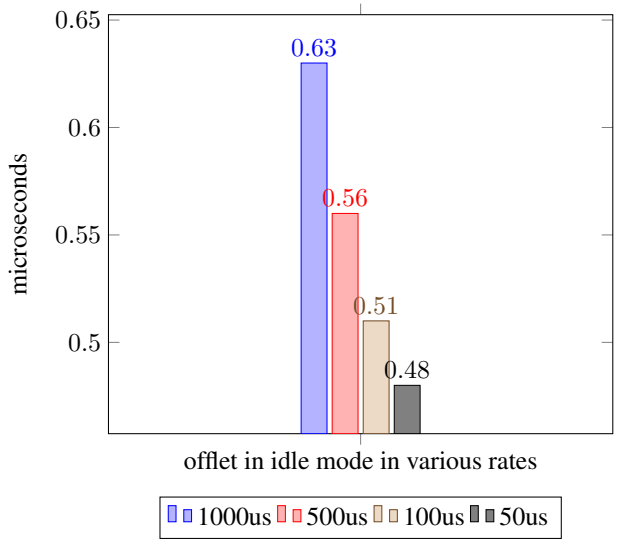


Figure 5 Maximum jitter in μs in idle mode

As noted, we toggle a GPIO for the measurements in EL1. From Figure 5, it appears that the transition from EL2 to EL1 and the GPIO toggle takes about 0.5 μs. Next, in Figure 6, we loaded the system by flooding the network card in order to examine the efficiency of the isolation.

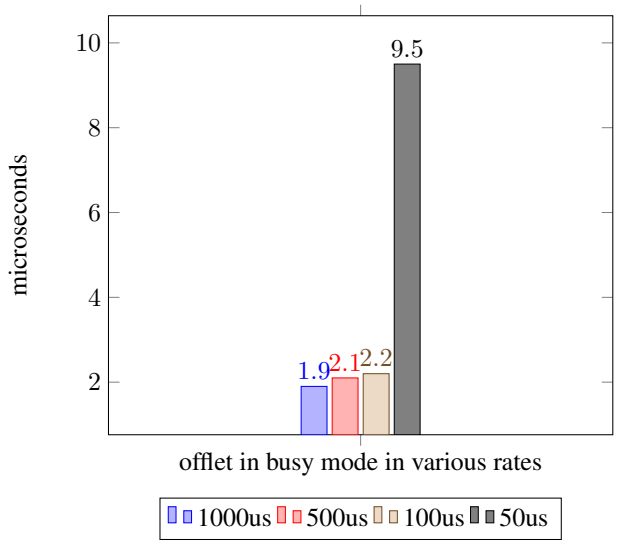


Figure 6 Maximum jitter in μs in busy mode

From Table 6 it is evident that a load in these frequencies is substantial. In the frequency of 20 kHz the worst case was 9.5 μs, which is nearly 20% of the cycle. The reason for the

jitter is the error of the PI's oscillator, which is 140 ppm and it is sensible that it is more observed at high frequencies. Table 2 presents some additional measures in busy mode.

Freq kHz	Avg	Stddev
1	1000165	170
2	500082	162
10	100016	150
20	50008	162

Table 2 Offline hyplet jitter (nanoseconds)

We see that bus sharing should be avoided as much as possible. Even if the process is entirely isolated from the GPOS there is still degradation in performance. The maximum jitter increases in general by 4 in low rates (up to 10 kHz). We also understand that the oscillator ppm deviation must be taken into consideration in high frequencies.

5.2 Ultrasonic distance

In this test, we used an ultrasonic sensor to measure the distance between the sensor (a consumer-grade HC-SR04) and an object. The sensor reports when the ultrasonic signal hits the echo sensor. Thus, by measuring the time elapsed between sending and receiving an ultrasound signal, we can deduce the distance. For this reason, it is important to measure the time as accurately as possible. Figure 7 shows an object located 30 cm from the sensor. The distance was chosen so that the duration would surpass 1 millisecond.

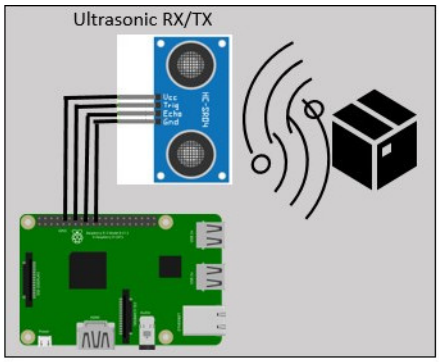


Figure 7 Ultrasonic schema

The speed of an ultrasonic sound wave is $34,300 \frac{cm}{s}$. Thus, assign the following:
Δt as the time elapsed between triggering and receiving.
d as the distance from the sensor to the object.
The calculation of d is:
$d = \Delta t \times 34,300/2$
To measure a distance with a sensor, we applied this relation in Algorithm 4, and implemented a program as an offline hyplet and as a native C user-space code.

Algorithm 4: ***Ultrasonic distance algorithm***

```
GPIO.Output(GPIO_TRIGGER) = True
Sleep 100 us
GPIO.Output(GPIO_TRIGGER) = False
Transmit for a 100us
StopTime = StartTime = time()
Take start time
while(GPIO.input(GPIO_ECHO) == 0) {
    StartTime = time()
}
When the echo GPIO is 1 the transmission began
while (GPIO.input(GPIO_ECHO) == 1) {
    StopTime = time()
}
When ECHO is 0 again then the ECHO signal is received completely
TimeElapsed = StopTime - StartTime
distance = (TimeElapsed * 34300) / 2
```

Table 3 displays the results of the native C program in RT_PREEMPT and in a hyplet mode. In this experiment, we did not impose any CPU load.

Test	Average	Stdev	Min	Max
RT_PRPT	30.2	1.2	29.2	32.4
hyp-offlet	30.4	0.04	30.3	30.4

Table 3 Ultrasonic, idle mode (in cm)

Now, we repeat the test but with disturbances. As in the timer test, we generate a large number of interrupts while performing the test ($3000\frac{INTR}{sec}$). Table 4 shows the averages and standard deviations.

Test	Average	Stdev	Min	Max
RT_PRPT	30.8	1.86	27.4	32.6
hyp-offlet	30.4	0	30.4	30.4

Table 4 Ultrasonic, busy mode (in cm)

The offset from 30 cm is probably due to the HCR sensor itself. Its oscillator is 40 kHz (25 μs), so the expected deviation is at most $0.000025 \times 343 = 0.00875 \approx 9mm$.

To summarise this test, the Offline Nanovisor overcomes RT_PREEMPT.

5.3 Infrared sensor

An infrared sensor setup is composed of a light trigger and an echo object. Usually, the sensor is connected to a GPIO , and the program reads it constantly.

To evaluate the accuracy of the Offline Nanovisor, we designed the experiment depicted in Figure 8. A beam is projected to a photo-resistor connected to a GPIO pin in a Raspberry Pi, which reads the digital value of the photo-resistor. In this experiment, we measured the interval between the moment we turn on the beam and the moment the photo-resistor raises the GPIO input to 'high'. We placed the beam projector 2 mm from the resistor. Light travels nearly at $300mm$ in 1 nanosecond. So, in theory we should have seen numbers less than one nanosecond; however, the actual numbers we got are much higher due to the PI's microcontroller delay. But we do see that the standard deviation differs.

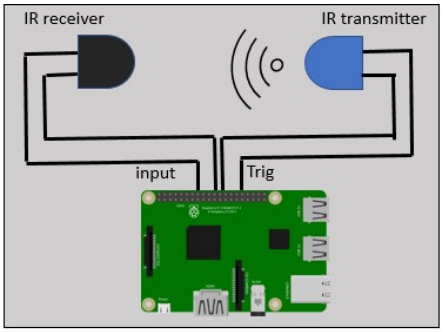


Figure 8 Infrared schema

Table 5 presents the difference between the native C program and its hyplet variant.

Test	Average	Stdev	Min	Max
RT_PRMPT	2077	20.5	2118	2056
hyp-offlet	2122	0.5	2122	2123

Table 5 Infrared (in μs)

Again, it is evident that the offline hyplet's programs are more accurate than native programs.

6 Related Work

Many groups have researched real-time operating systems (RTOS). Under the ARM architecture, other than RT_PREEMPT, we can find Jailhouse Baryshnikov [2016], Xen Barham et al. [2003], seL4 and OKL4 Heiser and Leslie [2010], Xenomai and RTAI, and many others. Of the six, okL4 is a closed-source microvisor not available to us and we do not discuss it.
As noted earlier, this paper does not claim that hyplets or the Offline Nanovisor are kernels but rather a Nanovisor extension to the GPOS. Moreover, the Offline Nanovisor may share the same computer with other technologies, such as seL4, as long as they do not use the same core. Nevertheless, we do provide some overview of related technologies, starting

with the microkernel. A microkernel is defined as the minimum set of functionalities needed to implement an operating system. Jochen Liedtke best describes it: 'A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations would prevent the implementation of the system's required functionality'. SeL4 is a microkernel for ARM8, ARM7, ARM6 and x86, and in some cases it is implemented as a microvisor. SeL4 architecture evolved from Liedtke's L4 microkernels family Elphinstone and Heiser [2013]. SeL4 provides a minimal set of functions to access a physical address space, interrupts, and processing time. It is also considered to be the only formally verified operating system Klein et al. [2009]. In general, seL4 is considered the fastest microkernel. Its support for SMP is considered experimental. SeL4 offers real-time Blackham et al. [2011] and security Sewell et al. [2011]. The weak part of seL4 is, however, its assimilation to existing hardware. It requires mastering CAmkES Kuz et al. [2007], a software component for microkernel-based embedded systems and a framework to build an operating system.

Jailhouse is a microvisor created by Siemens™. It uses partitioning to split the hardware into isolated compartments, or 'cells'. These cells are dedicated to executing programs called inmates Jailhouse does not emulate any device. Devices and processors are statically assigned to Jailhouse on creation. Unfortunately, Jailhouse, like KVM, does not run on Raspberry PI3. In Jailhouse, a single processor is assigned to perform the hard real-time tasks while the other processors are assigned to run Linux. Jailhouse is implemented as a Linux kernel driver and is a type-2 hypervisor, like the hyplets. Sebou Soltesz et al. [2007] show that Jailhouse's performance surpasses Xen's. This is mainly due to its simple design.

Other known open-source real-time operating systems for Linux, are Xenomai Gerum [2004] and RTAI Mantegazza et al. [2000]. Both technologies employ a microkernel architecture, meaning that the Linux kernel is merely a background task. Both technologies run on most processor architectures, e.g. x86, ARM7 and Power ISA. According to Barham et al. [2003], these two technologies perform somewhat the same while RTAI is a bit faster. However, as noted earlier, RTAI is not available on Raspberry PI and, as such, we feel that its development lags.

Dune Belay et al. [2012] is a system that provides a process rather than a machine, an abstraction through virtualisation. Dune offers a sandbox for untrusted code, a privilege separation facility and a garbage collector and is implemented on the Intel architecture. It is intended more for security than for RTOS. We believe we can port hyplets to Dune for ARMv8-a and ARMv7-a.

Rump kernels Kantee et al. [2012] are virtual lightweight containers for drivers in NetBSD Mewburn [2001]. Rump kernels run on top of the hypervisor and are processes running in hypervisor mode, and wrapped by containers that enable driver operations such as threads and synchronisation primitives. Rump kernels are designed for running drivers with little if any modification and still leave the kernel monolithic.

The Extended Berkeley Packet Filter, also known as eBPF **?** Borkmann [2016] is described as an in-kernel virtual machine that provides the ability to attach a program to a certain tracepoint in the kernel. Whenever the kernel reaches the tracepoint, the program is executed without a context switch. eBPF is undergoing massive development and is mainly used for packet inspection, tracing and probing. EBPF supports x86 architectures and ARM (although we failed to compile eBPF for ARMv8-a). It runs in kernel mode, which is considered unsafe, but uses a verifier to check for illegal accesses to kernel areas or the tampering of registers. Access to the user-space is done through memory maps.

EBPF uses LLVM, requires clang to generate a JIT code and has a quite small instruction set.

As a consequence, eBPF has substantial limitations as only a subset of the C language can be compiled into eBPF. EBPF has no loops, no native assembly, no static variables, no atomics, may not take a long time and is restricted to 4,096 instructions. Each eBPF instruction is 64-bit, so the biggest eBPF program may reach the size 4096 * 8 = 32 KB. But this is a byte code. Thus, in addition to the program size, there is also the overhead of LLVM. The hyplet runs native opcodes. Numerous vulnerabilities in an eBPF program might jeopardize the operating system. This is not the case with hyplets. The hyplet is not a program that executes in the kernel's address space but in the user's address space. Hence, there is no need for maps to share data between the user and the kernel. Hyplets do not require any particular compiler extensions, are much less restricted (what mapped prematurely can be accessed). Hyplets are meant to propagate events to a user-space program and process them in real-time, not just collect data as in eBPF.

7 Summary

The approach of embedding two distinct instances of two or more operating systems in a single computer has many justifications. However, this approach has some flaws. First, these operating systems require the programmers to master at least two operating systems: Linux and the microkernel's RTOS. The learnability of an operating system, its maintainability and portability are what make RT_PREEMPT so popular. RT_PREEMPT is easy and it is Linux. Just to shed some light, the RT_PREEMPT patch was evaluated (and thus promoted) by Cerqueira and Brandenburg [2013]; Regnier et al. [2008]; Arthur et al. [2007]; Mossige et al. [2007] , and is known today as the operating system taught in universities McLoughlin and Aendenroomer [2007].
Second, when we embed two operating systems on the same machine, we do that for a certain purpose. For instance, to have a GPOS that can run a GUI (graphical user interface) program that consumes data from the RTOS. Therefore some form of communication between the two operating systems is required. The communication between the two operating system needs to be maintained and synchronized. Thus, programmers need to be careful that it does not jeopardize the RTOS responsiveness. The Offline Nanovisor is intended to ease these challenges.

References

M. Ekman, F. Dahgren, P. Stenstrom, Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors, in: Proceedings of the international symposium on Low power electronics and design, IEEE, 2002, pp. 243–246.

M. Bennett, N. C. Audsley, Predictable and efficient virtual addressing for safety-critical real-time systems, in: Proceedings 13th Euromicro Conference on Real-Time Systems, IEEE, 2001, pp. 183–190.

R. Ben-Yehuda, Y. Wiseman, The offline scheduler for embedded vehicular systems, International Journal of Vehicle Information and Communication Systems 3 (2013) 44–57.

R. Ben Yehuda, N. Zaidenberg, Hyplets-multi exception level kernel towards linux rtos, in: Proceedings of the 11th ACM International Systems and Storage Conference, ACM, 2018, pp. 116–117.

W. Betz, M. Cereia, I. C. Bertolotti, Experimental evaluation of the linux rt patch for real-time applications, in: Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on, IEEE, 2009, pp. 1–4.

D. P. Bovet, M. Cesati, Understanding the Linux Kernel: from I/O ports to process management, " O'Reilly Media, Inc.", 2005.

M. Shalan, D. El-Sissy, Online power management using dvfs for rtos, in: Design and Test Workshop (IDT), 2009 4th International, IEEE, 2009, pp. 1–6.

A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, et al., Jitsu: Just-in-time summoning of unikernels., in: NSDI, 2015, pp. 559–573.

S. K. Datta, C. Bonnet, N. Nikaein, Android power management: Current and future trends, in: Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on, IEEE, 2012, pp. 48–53.

K. Paul, T. K. Kundu, Android on mobile devices: An energy perspective, in: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, IEEE, 2010, pp. 2421–2426.

N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, K. De Bosschere, Formal virtualization requirements for the arm architecture, J. Syst. Archit. 59 (2013) 144–154.

C. Dall, J. Nieh, Kvm/arm: The design and implementation of the linux arm hypervisor, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, ACM, New York, NY, USA, 2014, pp. 333–348.

A. Resh, N. Zaidenberg, Can keys be hidden inside the cpu on modern windows host, in: Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013, Academic Conferences Limited, 2013, p. 231.

R. B. Yehuda, N. J. Zaidenberg, Protection against reverse engineering in arm, International Journal of Information Security (2019) 1–13.

S. Rostedt, D. V. Hart, Internals of the rt patch, in: Proceedings of the Linux symposium, volume 2, 2007, pp. 161–172.

B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, G. Heiser, Timing analysis of a protected operating system kernel, in: Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd, IEEE, 2011, pp. 339–348.

A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, C. Kozyrakis, Dune: Safe user-level access to privileged cpu features., in: Osdi, volume 12, 2012, pp. 335–348.

A. Kantee, et al., Flexible operating system internals: the design and implementation of the anykernel and rump kernels (2012).

P. Regnier, G. Lima, L. Barreto, Evaluation of interrupt handling timeliness in real-time linux operating systems, ACM SIGOPS Operating Systems Review 42 (2008) 52–63.

D. A. Fitriani, W. Andhyka, D. Risqiwati, Design of monitoring system step walking with mpu6050 sensor based android, JOINCS (Journal of Informatics, Network, and Computer Science) 1 (2017) 1–8.

X. Wang, M. Mizuno, M. Neilsen, X. Ou, S. R. Rajagopalan, W. G. Boldwin, B. Phillips, Secure rtos architecture for building automation, in: Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy, ACM, 2015, pp. 79–90.

A. Patel, M. Daftedar, M. Shalan, M. W. El-Kharashi, Embedded hypervisor xvisor: A comparative analysis, in: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2015, pp. 682–691.

T. N. B. Anh, S.-L. Tan, Real-time operating systems for small microcontrollers, IEEE micro 29 (2009) 30–45.

M. Baryshnikov, Jailhouse hypervisor, B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: ACM SIGOPS operating systems review, volume 37, ACM, 2003, pp. 164–177.

G. Heiser, B. 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.

K. Elphinstone, G. 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, ACM, 2013, pp. 133–150.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., sel4: Formal verification of an os kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 207–220.

T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, G. Klein, sel4 enforces integrity, in: International Conference on Interactive Theorem Proving, Springer, 2011, pp. 325–340.

I. Kuz, Y. Liu, I. Gorton, G. Heiser, Camkes: A component model for secure microkernel-based embedded systems, Journal of Systems and Software 80 (2007) 687–699.

S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, in: ACM SIGOPS Operating Systems Review, volume 41, ACM, 2007, pp. 275–287.

P. Gerum, Xenomai-implementing a rtos emulation framework on gnu/linux, White Paper, Xenomai (2004) 1–12.

P. Mantegazza, E. Dozio, S. Papacharalambous, Rtai: Real time application interface, Linux Journal 2000 (2000) 10.

L. Mewburn, The design and implementation of the netbsd rc. d system., in: USENIX Annual Technical Conference, FREENIX Track, 2001, pp. 69–79.

Creating complex network services with ebpf: Experience and lessons learned, Miano, Sebastiano and Bertrone, Matteo and Risso, Fulvio and Tumolo, Massimo and Bernal, Mauricio Vásquez, 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), 1–8, 2018, IEEE.

D. Borkmann, On getting tc classifier fully programmable with cls bpf., tc (2016).

F. Cerqueira, B. Brandenburg, A comparison of scheduling latency in linux, preempt-rt, and litmus rt, in: 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, SYSGO AG, 2013, pp. 19–29.

S. Arthur, C. Emde, N. Mc Guire, Assessment of the realtime preemption patches (rt-preempt) and their impact on the general purpose performance of the system, in: Proceedings of the 9th Real-Time Linux Workshop, 2007.

M. Mossige, P. Sampath, R. Rao, Evaluation of linux rt-preempt for embedded industrial devices for automation and power technologies-a case study, in: 9th RTL Workshop [Online]. Available: http://www. linuxdevices. com/files/article081/Sampath. pdf, 2007.

I. McLoughlin, A. Aendenroomer, Linux as a teaching aid for embedded systems, in: Parallel and Distributed Systems, 2007 International Conference on, volume 2, Ieee, 2007, pp. 1–8.

Raz Ben-Yehuda and Nezer Zaidenberg, The hyplet - Joining a Program and a Nanovisor for real-time and Performance, SPECTS, The 2020 International Symposium on Performance Evaluation of Computer and Telecommunication Systems Conference, 2020

PX

C FLAT NANOVISED

by

Raz Ben Yehuda, Adam Aronov, Or Ekstein, Michael Kiperberg and Nezer Zaidenberg

submitted

NANOVISED C-FLAT Linux

JDMS
XX(X):1–11

DOI: 10.1177/xxx
www.sagepub.com/
SAGE

Raz Ben Yehuda [1] Michael Kiperberg [2] Adam Aronov [3] Or Ekstein [3] and Nezer Jacob Zaidenberg[1,3]

Abstract

This paper presents the implementation of control flow attestation (C-FLAT) for Linux. C-FLAT is a control attestation system for embedded systems, and was implemented in ARM through TrustZone on bare-metal devices. We extend the design and implementation of C-FLAT through the use of a type 2 Nanovisor in the Linux operating system. Compared to the original C-FLAT, C-FLAT Linux reduces processing overhead and is able to detect a SlowLotis attack. We detail the architecture and offer benchmarks, real test cases, and demonstrate detection of a SlowLoris attack on the Apache webserver.

Keywords

ARM, Virtualization, Hypervisor, CFI

Introduction

C-FLAT by Abera et al. Abera (2016) is a technique for attesting an application's control flow on an embedded device. C-FLAT is a dynamic analysis tool. It complements static attestation by capturing the program's runtime behavior and verifies the exact sequence of executed instructions, including branches and function returns. It allows the verifier to trace the program's flow control to determine whether the application's control flow was compromised. Combined with static attestation, C-FLAT can precisely attest embedded software execution.

Originally, C-FLAT design allows attestation for simple systems. C-FLAT does not support threads, processes or operating systems. Therefore, most industrial systems are too complex for C-FLAT. Complex systems usually require multi-processing, multi-threading, inheritance or function pointers etc. These features are available on a General Purpose Operating System (GPOS), C-FLAT does not support GPOS.

Furthermore, C-FLAT runs on top of TrustZone. TrustZone programming requires high expertise as well as access to the boot loader code. Such access is usually available only to the SoM (System On a Module) vendor. Here, we provide a similar but more straightforward approach by using our a dedicated Nanovisor yehuda (2020). In this paper, we extend C-FLAT and eliminate the following limitations:

- **Single threads**. C-FLAT is available only for a single thread.
- **Single Process**. C-FLAT is available only for a single process.
- **Multi-core** C-FLAT is utilized only for a single process.
- **TrustZone access** C-FLAT requires access to TrustZone Flur (2016); Tice (2014), which may not be available in many industrial cases.

The paper's main contribution is adapting C-FLAT for complex applications in a GPOS (Linux). We demonstrate how our system can detect real exploits such as SlowLoris, that affect production systems, and handles a real test case (CVE-2019-9210).

We record the control flow path and send continuous subsequences to an attestation server. Furthermore, as a result of using Linux, the attestation server may execute locally. Local execution reduces the risk of transmitting the data over the network and allows for better performance.

Also, as we use C-FLAT over a complex general-purpose application, we must carefully approach the performance penalty. For this reason, we offer a novel approach to mitigate it. We run the instrumented section alternately. For example, execute this code section once a second.

Our contribution provides software-only CFI for ARM for software running in a GPOS. We perform the path monitoring in TEE or HYP mode. We provide a context-sensitive CFI and is available for 32bit applications, and is useful on Android Phones. Our Nanovisor software can run on EL2 (HYP mode) or TrustZone. Since it can track suspicious paths, it helps in detecting DOS attacks.

Related work

Research in CFI is vast and is available in the hardware and the software.

ARMv8.3-PAuth Flur (2016) adds PAC (Pointer Authentication Code). It validates whether the target of an indirect branch is correct (Figure 1).

[1] University of Jyväskylä, Finland
[2] SCE, Israel
[3] College of management academic studies, Israel

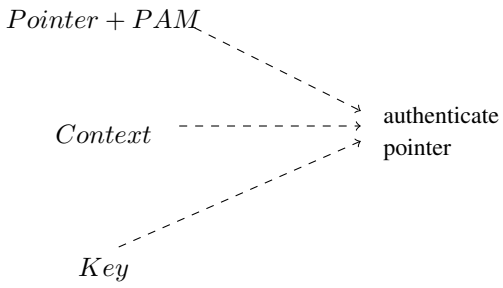


Figure 1. PAC

ARMv8.3 hardware is not available at the time of writing. Also, the authentication is done in EL0.

Clang CFI Abadi (2009) is designed to detect schemes of undefined behavior in C++ programs. These schemes have been optimized for performance. The schemes rely on LTO (Link Time Optimization). For better efficiently, the program must be structured such that certain object files are compiled with CFI enabled. The schemes focus on casting between types, incorrect virtual calls and indirect virtual calls. It can be interesting to perform Clang's CFI in our Nanovisor.

There are several control flow attestation systems similar to C-FLAT, all having similar limitations. Lo-FatDessouky (2017) is similar system for RISC-V architecture. A similar system for the bare metal system was proposed in Clements (2017). Our advantage against all of these is the ability to support GPOS and normal multiprocess, and multi-threaded applications.

Kernel CFI (kCFI) Moreira (2017) demonstrates CFI for the Linux kernel. It combines static analysis at the source and binary levels, and creates a restrictive CFI policy. Compared to other kernel CFI approaches, it achieves a small overhead, approximately 2%, and supports dynamic module insertion. kFCI is compiler-based technology and, therefore does not rely on any runtime supervisor or routine, avoiding overheads.

MoCFI Davi (2012) (Mobile CFI) is a CFI framework intended for ARM7. It instruments the binary on the fly and offers protection from code injection attacks, and code reuse attacks (such as return-to-libc). Unlike C-FLAT the attestation is performed in a non TEE environment, and the binary analysis is done offline. However, it is only fair to say that it is possible to migrate the runtime enforcement into our Nanovisor.

PathArmour Van (2015) is a context-sensitive CFI (CCFI) for the Intel platform. Its uniqueness is its ability to detect wrong code path flows, for example, if function B can be called from both A and C, but the current context is A to B but the actual path is C to B, then PathArmour can detect this. PathArmour uses a kernel module to monitor paths. To reduce the overhead it mainly focuses on system call tracing and relies on x86_64 branch recording features. Unlike C-FLAT, it does not run in ARM and the path monitoring is not performed in a TEE.

Another technology for ROP attacks is the kBouncer Pappas (2012). KBouncer is a binary CFI for the x86 platforms. it focuses on ROP attacks protection, through the use of Intel's LBR (Last Branch Recording). LBR is a set of registers that record the last branches, it is transparent to the running application and therefore incurs zero overhead for storing the branches.

In the area of static attestation, we find SWATT (Software-based Attestation Technique) Seshadri (2004) for embedded devices. SWATT examines the memory content for viruses. Another software is this area is Pioneer Seshadri (2005). In Pioneer the executable is guaranteed to execute a trusted environment by implementing a root of trust. This dynamic root of trust is instantiated through the verification function, a self-checking function that checksums its instructions. Viper Li (2011) verifies the integrity of peripherals firmware. SMART Eldefrawy (2012) is a small modification to a micro-controller that is used to facilitate a dynamic root of trust in remote embedded devices. TrustLite Koeberl (2014) is an FPGA to enforce software modules protection. Another FPGA solution is TyTan Brassers (2015), a trust anchor for tiny embedded that provides secured task loading, secured IPC and local and remote attestation.

CFI performance and efficiency had been researched heavily in the past years Tice (2014); Wei (2013); Zhang (2013). CPI (Control Pointer Integrity) Kuznetsov (2014); Evans (2015) is a different technique for insuring pointers in the code. It protects from code hijack but does not provide information about control flow paths. Property-based attestation Nagarajan (2009) however, Sadegi (2004) Chen (2006) demonstrated some deficiencies, to name a few, it reveals information about the platform's configuration(hardware and software) and the application, usually through an external attesting agent. Besides, all trusted permutations of the trusted configurations must be known beforehand. Also, if the configuration changes, this must reflect a verifier. Software attestation calculates a hash over the program's code, and the correctness relies also on that the verifier responds in time. This real-time requirement may not be possible in a busy network.

C-FLAT and our approach to attestation is dynamic analysis. Using hypervisors for dynamic security of kernels is described in SecvisorSeshadri (2007). Secvisor can assure that only whitelisted code executes. However, Secvisor does not protect against return-oriented programming or perform dynamic analysis. Liu (2018) et al. describe KSP (Kernel Stack Protection), through the use of a hypervisor. In Liu (2018), a return-to-schedule attack is made on some process while this process not executing and its stack is tampered. The protection model is by shadow page tables. These pages are used to provide different kernel stacks with different access permissions; thus, kernel units have different access permissions when they try to modify other kernel stacks; in addition Liu (2018) et al. offer to use the hypervisor to record important information regarding the process, and thereby protecting from some malicious kernel. MOSKG (Multiple Operating Systems Kernel Guard) is aimed to protect from DKOM attacks (Dynamic Kernel Object Manipulation), Yan (2015) et al. offers a secure paging mechanism to protect critical kernel data.

HIMAAzab (2009) is a similar system for detecting kernel integrity using a hypervisor. We performed a dynamic analysis of kernel code using Lguest in Khen (2013). The method was extended Zaidenberg (2015) for detecting operating system bugs and kernel vulnerabilities. Lgdb is

based on a para-virtualized environment and significant performance penalties. It is limited to a single version of Linux (running on the host) and 32bit x86. In contrast, our new approach is using hardware virtualization with good performance. Our approach is aimed directly at detecting ROP (Return Oriented Programming) in real-time. Our new approach is not limited to Linux of any flavor, provides much better performance and uses ARM hardware-assisted virtualization.

Another approach to dynamic analysis is the phases approach. On the first (online) step, the entire memory of the inspected system is grabbedKiperbeg (2019). Then the memory is examined using volatilityLigh (2014) or AI-based tools to detect anomalies in memory Case (2017). The problem of live memory forensics is that the comparable memory is large. Therefore, changes occur while the forensic analysis is running causing anomalies Aljaedi (2011). Unlike our method, independent memory acquisition and analysis are not capable of understanding context and are vulnerable to detection anomalies. In contrast, we trace the stack of the system and are aware of loading libraries and performing function and system calls.

Threat Model

C-FLAT does not target specific attacks (ROP, MOSKG ,DOS) attacks or similar attacks, therefore it is difficult to compare C-FLAT to other technologies. C-FLAT targets vulnerabilities once a zero-day attack takes place. C-FLAT, due its versatile nature, also records the time of execution, and therefore stipulates better diagnostics of inconsistencies in the time of execution in the flow of the program.

For this paper, an attacker may attack via the network through DOS attacks, or in the case of CFI attacks, the attacker may change the program's memory by exploiting some vulnerability Lu (2019). In this paper, a CFI attack refers to an attack on a native binary. For DOS attacks, we assume that the HTTP request bypassed any network firewall if one exists.

Background

C-FLAT is a dynamic attestation and complements static attestation. It measures the program's execution path at the opcode level by capturing its runtime behavior. Figure 2 (taken from the original C-FLAT paper) presents the C-FLAT. *Prv* is a prover, and *Ver* is the verifier. The Prover usually runs on a resource-limited device, and therefore cannot do the verification while in execution. Both the Verifier and the Prover must have access to the program's binary at any time.

First step ① is that *Ver* generates a control-flow-graph offline. It also measures each possible control-flow path using function *H*, and saves the results ②. In the original C-FLAT, *Ver* stores and generates CFG's results in *Ver*, because *Prv* is a low resources' computer. In C-FLAT Linux the CFG is stored in *Ver* itself. Reading the CFG is required only once per program. We do it on the process start-up, so it doesn't dangers the performance. In ③, *Ver* challenges *Prv*. Then, *Prv* starts executing the program ④, while our trusted software computes the program's path ⑤. In the original C-FLAT, code running in TrustZone performed this computation. In our system code, running in hypervisor mode performs the computation. Lastly, *Prv* generates the attestation report $r = SigK(Auth, c)$, computed as a digital signature over the challenge c and Auth using a key K known only to the measurement engine. Lastly, *Prv* transmits r to *Ver* ⑥ to validate it ⑦.

C-FLAT model is designed for small embedded devices, and therefore does not fit to a general-purpose OS on a general-purpose processor which is common today in many embedded setups. C-FLAT Linux is suitable for these setups.

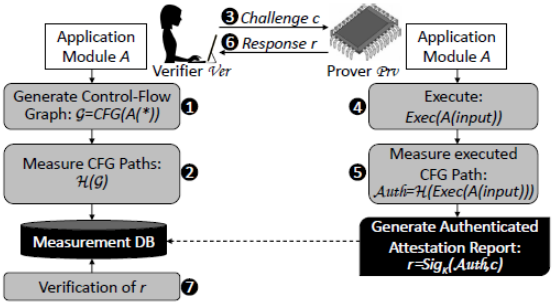


Figure 2. C-FLAT original system model ©Abera

In addition to the above, in-order for C-FLAT to trace the program's flow, it is essential to intervene in the program binary (Original). For this, we replace the program's branches and returns-from-function. For example, in Figure 3, the "br x0" opcode is replaced by "br hook_b" opcode (Instrumented). Hook_b is an address of a procedure that searches for the address of the original branch opcode, and once found (Trampoline), it sends (via the SMC command) it to computation in the TrustZone. The computation records the address and returns to perform the original branch command.

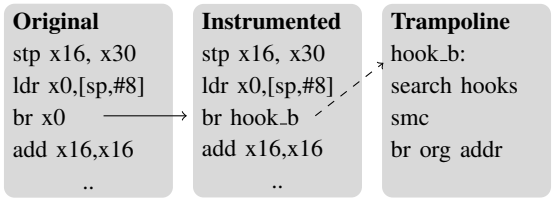


Figure 3. C-FLAT Instrumented

Control flow attestation for Linux

Our Nanovisor implementation (Figure 4) replaces the TrustZone. C-FLAT Linux instruments the ELF Lu (1995) (Executable Linking Format) much like in Abera (2016). The instrumentation includes replacing the binary opcodes for the various "branch" commands to a code that branches to the trampoline. It also requires caching the original branch targets.

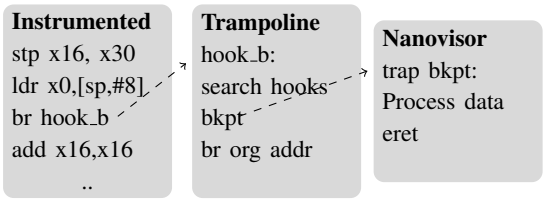


Figure 4. A C-FLAT Program in Linux

Figures 4 and 3 depict the main difference in the infrastructures. Our Nanovisor provides a trusted execution environment (TEE) through a fast Nanovisor RPC yehuda (2020). The Nanovisor performs the computation and aggregation of addresses generated by the program's flow.
Figure 5 depicts that as the instrumented program P runs, it generates input to Nanovisor Q. Q collects the input, and at some point, the collected data is encrypted and passed to the attestation server. The attestation server deciphers the data and verifies that the control path does not have any abnormalities. The attestation server handles each branch instruction or batches of instructions.

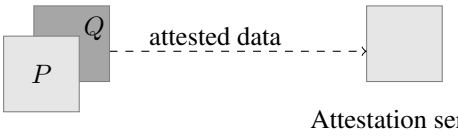


Figure 5. C-FLAT Architecture

Program Q (Figure 5) is a background process that serves the Nanovisor requests from the protected program. Parts of Q execute in HYP mode through the use of our Nanovisor RPC yehuda (2020). Q loads when the computer boots and waits in the background. C-FLAT does not start any threads or agents, and its overhead is small. Q's main task is to process the information from the protected process and pass it to the attestation server. The information that the protected program passes Q is the Event Type, the Destination address, the Source address and Link Register Value. These values are passed on four registers to increase performance. But, the constant round trip of the traps incurs an overhead. We will demonstrate the overhead in the evaluation section. To improve performance we decided to mitigate the number of round-trips.

Performance

As we aim to mitigate the traps penalty, we need to minimize Nanovisor traps' (calls) amount. Thus, we trigger off and on the trap code. To do that, we intervene in the protected program execution code while it executes, and replace the trap opcode (the BKPT instruction) with the NOP (no operation) opcode. A NOP opcode depends on the compiler output, and therefore to modify the opcode in real-time, it is required to know in advance the following

1. The size of the BKPT opcode (16 bit or 32 bit).
2. The position of the opcode in the program's address space
3. Identify that a protected program runs (or sleeps) without the need to wait for traps.

The ARM compiler may generate opcode in thumb mode (16bit) or regular mode (32bit). For this reason, it is essential to know the BKPT (breakpoint) opcode size and position. Therefore, the Nanovisor records the position (and size) of the first trap generated by the INIT operation (Figure 6). The INIT operation is also used to initialize C-FLAT's background process (Q). At some early point in the program execution, the page containing the trap code is mapped to the kernel. The user (which may be a program), at his choice, triggers the coding and re-coding of the BRPT/NOP opcodes.

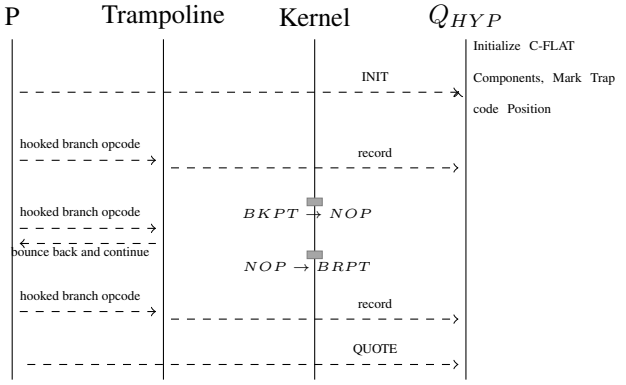


Figure 6. C-FLAT Flow

The trap to the Nanovisor is performed by programming the Nanovisor to trap access to BKPT by setting the mdcr_el2 register (breakpoint control register). This means that any program that triggers a breakpoint enters the Nanovisor. Since there are systems that require debug infrastructure, only when the protected program executes the trap is set, and after the scheduler switches the process, it is unset. C-FLAT framework names the protected ELF sections ".attest". Each program invocation triggers a scan for the ".attest" ELF sections. Then the Nanovisor enables the trap for the protected process. Thus, any other process that triggers a breakpoint does not enter the Nanovisor.
To summarize, when possible, we can set C-FLAT to execute arbitrarily, thus improving performance and reducing processor cycles and heat.

GPOS considerations

Original C-FLAT run on an embedded ARM-based system without GPOS. In a GPOS Environment, there are additional considerations. First of which is Multi-core. Linux supports Multi-core computers. In Multi-core architectures, a process or a thread might migrate among cores. Thus on multi-core systems, our Nanovisor must execute on all cores.
Figure 7 demonstrates how multiple C-FLAT threads of the same process run concurrently. Therefore, to solve the concurrency problem, we set on each processor's core a C-FLAT Nanovisor that accesses a single instance of a process Q.

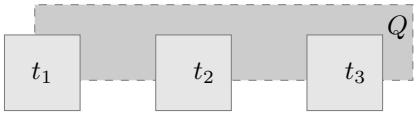


Figure 7. C-FLAT Multi-core Server

Another facet of GPOS is context switching. Assuming that a protected program P is infected by a virus, and thereby it refrains access to the Nanovisor. When C_FLAT is executing in a non GPOS, it is likely, that most if not all of the time, a single process occupy the processor. Therefore, after some time that C-FLAT did not execute, the system may be infected by a virus. In a GPOS, we cannot make that assumption, as other processes may preempt the protected program, or that the protected program is not executing. For that, we need to know that a C-FLAT process executes, and when it does, we expect C-FLAT to trigger shortly. For this reason, in each context switch, we also check whether the C-FLAT program enters or leaves the processor, and record the time and the traps count.
Multi-threading also challenges C-FLAT because threads are created and destroyed ad-hoc. We, therefore, offer that each thread state is kept in a table entry in the C-FLAT server. Once a thread is destroyed, we mark this entry as free. We use the TLS (thread-local storage) Drepper (2003) to differentiate between the different threads and processes. The Linux kernel in ARM puts on the tpidr_el0 register the TLS value, which is accessible in the Nanovisor.
Multiple C-FLAT processes are possible. Multi-processes design is similar to the multiple threads design we discussed.

The hyplet Nanovisor

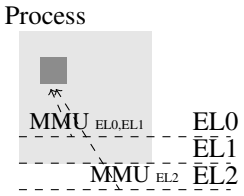
Here we describe the Nanovisor technology that we referred to as the hyplet yehuda (2018, 2020). ARM8v-a specifications offer to distinct between user-space addresses and kernel space addresses by the MSB (most significant bits). The user-space addresses of Normal World and the hypervisor use the same format of addresses.
These unique characteristics are what make the hyplet possible. The Nanovisor can execute user-space position-independent code without preparations. Consider the code snippet at Figure 8. The ARM hypervisor can access this code's relative addresses (adrp), stack (sp_el0) etcetera without pre-processing. From the Nanovisor perspective, Figure 8 is a native code. Here, for example, address 0x400000 is used both by the Nanovisor and the user.

```
400610: foo:
400614: stp x16, x30, [sp,#-16]!
400618: adrp x16, 0x41161c
40061c: ldr x0, [sp,#8]
400620: add x16, x16, 0xba8
400624: br x17
400628: ret
```

Figure 8. A simple hyplet

So, if we map part of a Linux process code and data to a Nanovisor it can be executed by it.

To make sure that the program code and data are always accessible and resident, it is essential to disable evacuation of the program's translation table and cache from the processor. Therefore, we chose to constantly accommodate (cache) the code and data in the hypervisor translation registers in EL2 cache and TLB. To map the user-space program, we modified the Linux ARM-KVM Dall (2014) mappings infrastructure to map a user-space code with kernel space data.



Two exception Levels access the same physical frame with the same virtual address of some process. However, the page tables of the two exception levels are not identical.

Figure 9. Asymmetric dual view

Figure 9 demonstrates how identical addresses may be mapped to the same virtual addresses in two separate exception levels. The dark shared section is part of EL2 and therefore accessible from EL2. However, when executing in EL2, EL1 data is not accessible without previous mapping to EL2. Figure 9 presents the leverage of a Linux process from two exception levels to three.

The hyplet protection

The natural way of memory mapping is that EL1 is responsible for EL1/EL0 memory tables and EL2 is responsible for its memory table, in the sense that each privileged exception level accesses its memory tables. However, this would have put the Nanovisor at risk, as it might overwrite or otherwise garble its page tables. As noted earlier, on ARM8v-a, The hypervisor has a single memory address space. (unlike TrustZone that has two, for kernel and user). The ARM architecture does not coerce an exception level to control its memory tables. This makes it possible to map EL2 page table in EL1. Therefore, only EL1 can manipulate the Nanovisor page tables. We refer to this hyplet architecture as a Non-VHE hyplet. Also, to further reduce the risk, we offer to run the hyplet in injection mode. Injection mode means that once the hyplet is mapped to EL2, the encapsulating process is removed from the operating system kernel, but its hyplet's pages are not released from the Nanovisor, and the kernel may not re-acquire them. It is similar to any dynamic kernel module insertion.
In processors that support **VHE** (Virtual Host Extension), EL2 has an additional translation table, that would map the kernel address space. In a VHE hyplet, it is possible to execute the hyplet in the user-space of EL2 without endangering the hypervisor. A hyplet of a Linux process in $EL0_{EL1}$ (EL0 is EL1 user-space) is mapped to $EL0_{EL2}$ (EL2 user-space). Also, the hyplet can't access EL2 page tables because the table is accessible only in the kernel mode of EL2. VHE resembles TrustZone as it has two distinct address spaces, user and kernel. Operating systems such as QSEE (Qualcomm Secure Execution Environment) and OP-TEE optee (2010) are accessed through an upcall and execute the user-space in TrustZone. Unfortunately, at the time of writing, only modern ARM boards offer VHE extension (ARMv8.2-a) and therefore this paper demonstrates benchmarks on older boards.

The hyplet security

As noted, VHE hardware is not available at the time of this writing, and as such we are forced to use software measures to protect the hypervisor. On older ARM boards it can be

argued that a security bug at hypervisor privilege levels may cause greater damages compared to a bug at the user process or kernel levels thus poising system risk.
The hyplet also escalates privilege levels, from exception level 0 (user mode) or 1 (OS mode) to exception level 2 (hypervisor mode). Since the hyplet executes in EL2, it has access to EL2 and EL1 special registers. For example, the hyplet has access to the level 1 exception vector. Therefore, it can be argued that the hyplet comes with security cost on processors that do not include ARM VHE.
The hyplet uses multiple exception levels and escalates privilege levels. So, it can be argued that using hyplets may damage application security. Against this claim, we have the following arguments.
We claim that this is risk is superficial and an acceptable risk, for processors without VHE support. Most embedded systems and mobile phones do not include a hypervisor and do not run multiple operating system.
In the case where no hypervisor is installed, code in EL1 (OS) has complete control of the machine. It does not have lesser access code running in EL2 since no EL2 hypervisor is present. Likewise code running in EL2 can affect all operating systems running under the hypervisor. Code running in EL1 can only affect the current operating system. When only one OS is running the two are identical.
Therefore, from the machine standpoint, code running in EL1 when EL2 is not present has similar access privileges to code running in EL2 with only one OS running, as in the hyplet use case.
The hyplet changes the system from a system that includes only EL0 and EL1 to a system that includes EL0, EL1, and EL2. The hyplet system moves a code that was running on EL1 without a hypervisor to EL2 with only one OS. Many real-time implementations move user code from EL0 to EL1. The hyplet moves it to EL2, however, this gains no extra permissions, running rogue code in EL1 with no EL2 is just as dangerous as moving code to EL2 within the hyplet system. Additionally, it is expected that the hyplet would be a signed code; otherwise, the hypervisor would not execute it.
The hypervisor can maintain a key to verify the signature and ensure that lower privilege level code cannot access the key.
Furthermore, Real-time systems may eliminate even user and kernel mode separation for minor performance gains. We argue that escalating privileges for real performance and Real-time capabilities is an acceptable on older hardware without VHE where hyplets might consist of a security risk. On current ARM architecture with VHE support the hyplet do not add extra risk.

Static analysis to eliminate security concerns

Most memory (including EL1 and EL2 MMUs and the hypervisor page tables) is not mapped to the hypervisor. The non-sensitive part of the calling process memory is mapped to EL2. The hyplet does not map (and thus has no access to) kernel-space code or data. Thus, the hyplet does not pose a threat of unintentional corrupting kernel's data or any other user process unless additional memory is mapped or EL1 registers are accessed.
Thus, it is sufficient to detect and prevent access to EL1 and EL2 registers to prevent rogue code affecting the OS memory from the hypervisor. We coded a static analyzer that prevents access to EL1 and EL2 registers and filters any special commands.
We borrowed this idea from eBPF Corbet (2018). The code analyzer scans the hyplet opcodes and checks that are no references to any black-listed registers or special commands. Except for the clock register and general-purpose registers, any other registers are not allowed. The hyplet framework prevents new mappings after the hyplet was scanned to prevent malicious code insertions. Another threat is the possibility of the insertion of a data pointer as its execution code (In the case of SIGBUS of SEGFAULT, the hyplet would abort, and the process terminates). To prevent this, we check that the hyplet's function pointer, when set, is in the executable section of the program.
Furthermore, the ARM architecture features the TrustZone mode that can monitor EL1 and EL2. The TrustZone may be configured to trap illegal access attempts to special registers and prevent any malicious tampering of these registers.
The Nanovisor is 768 lines of code. It includes the interrupt vector (300 lines) and the hyplet's user-kernel interface. It is part of the Linux kernel and therefore open-source. Its tiny size eases code analysis and its protection (ex. Hypersafe Wang (2010)). As noted, we re-used the Linux's KVM infrastructure. KVM is well-debugged, and thus it reduces risks of errors; Also, future development in the area of virtualization in Linux may be adapted to the hyplet.
Control Flow Attestation demands a facility for static remote attestation capable of attesting the instrumented code. Otherwise, the control flow might be easily spoofed by an adversary that adds or removes instrumentation or traps. Therefore, we suggest encrypting the ensuring function, as suggested in yehuda (2020) et al. . yehuda (2020) et al. presented a technique to sign digitally pieces of code (functions and data) through the use of the hyplet.

Evaluation

We provide benchmarks and compare our solution to Normal runs. We use Raspberry PI3 to demonstrate. The Raspberry PI3 main specifications are shown in Table 1:

Soc	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2 GHz, (clocked to 700 MHz)
RAM	1GB LPDDR2 (900 MHz)
Clock	19.2 Mhz

Table 1. PI3 specifications

We first start with evaluating the Nanovisor RPC (the trap), we then provide a synthetic benchmark and evaluate a simple instrumented procedure, evaluate a real test case CVE 2019-9210, and lastly, we show C-FLAT for Apache.

Access duration

To access the Nanovisor we set a trap on the BRK opcode from the trampoline to the attestation server. In the evaluated hardware, access through an "ioctl" operation to the kernel is approximately 500 nanoseconds yehuda (2020). Table 2 presents the time to move from user space to the Nanovisor.

Measure	BRK trap
Avg	92 ns
StdDev	41 ns
Max	156 ns
Min	52 ns

Table 2. BRK vs. SVC & Ioctl

Evidently, as seen from Table 2, on average, trap costs are quite small.

Performance

Here we measure the actual penalty of C-FLAT. We conducted this test in the following configurations:

- Native mode
  A native mode is when we execute without C-FLAT's instrumentation.
- Protected mode
  A protected mode is when we execute the protected sections with C-FLAT's instrumentation and access the Nanovisor.
- Non Protected mode
  A non-protected mode is when we execute the protected sections without accessing the Nanovisor. This means that the BKPT opcode is replaced by the NOP opcode, and the process accesses only the trampoline.

Simple test We first measure a CPU intensive program.

```
extern void do_odd(int x,int y);
extern void do_parity(int x,int y);

void foo(int loops)
{
    for(int i = 0; i < loops; i++){
        if ( i % 2 )
            do_odd(i,loops);
        else
            do_parity(i, loops);
    }
}
int main() {
    foo(100000);
}
```

The functions do_odd and do_parity multiply the two arguments and return the value. To get more accurate results, in Table 3 each test was performed 10 times.

Test	Avg	Max	Min	StdDev
Native	5188	5823	5055	241
NotProtected	37097	37237	36979	92
Protected	148056	154086	146994	2174

Table 3. 1000 iterations (μs)

The instrumentation slows down the program 7 times, and the penalty of entering the Nanovisor slows the program 28 times.

Analysis
We want to calculate the actual attestation penalty. Table 4 presents the the parts composing the flow. We wish to isolate the attest cost.

Native	Trampoline	Hyp Access	Attest

Table 4. C-FLAT CFI complete flow

The program jumps to the trampoline in two main positions:

1. The exit condition from the for loop: $i < loops$:
2. The decision of which function to invoke: i % 2:

For each of the above, the program calls the trampoline 100000 times, therefore, we reach over 200000 trampoline calls. We showed in Table 2 that the average cost of entering the Nanovisor is on average 92 nanoseconds, therefore, the Nanovisor penalty access is:
$92 \times 200000 = 18400000 = 18400\mu s$
The pure C-FLAT attestation processing can be calculated by subtracting from the total duration, the Native, the trampoline access,
$148056 - 5188 - 37097 - 18400 = 87371\mu s$.
Therefor, the attestion part takes about:

$$\frac{87}{148} = 0.58 \approx 60\%$$

of the total C-FLAT overhead.

Conclusion
C-FLAT's attestation algorithm is too time-consuming for CPU intensive programs. C-FLAT does not apply to these types of applications.

Test	Avg	Max	Min	Std dev
Native	1628	1688	1606	31.8
Protected	2655	2795	2625	52

Table 5. Good Input (ms)

Test	Avg	Max	Min	Std dev
Native	21.7	23	21	0.67
Protected	113	119	111	3

Table 6. Erroneous Input(ms)

Real Test case **Analysis**
From the above table 5 and table 6, it is evident that C-FLAT incurs an overhead. In the erroneous input, the overhead is 5 times slower, while in the good input it is 2 times slower. The reason is that the good input is much bigger (1/2 MB on average) than the erroneous input (200 bytes), and therefore, less I/O activity is involved. The standard deviation in Table 6 is 4.5 times bigger in the Protected mode, and in Table 5 it is 1.6 times bigger . We believe that this is because the program exits with an error (SEGFAULT). As a consequence

of the program's violation, the operating system performs other time-consuming activities, such as an I/O.

Conclusions
C-FLAT may be useful in some cases, such as I/O intensive programs.

SlowLoris attack

We used C-FLAT to detect a SlowLoris denial of service((dos) attack Damon (2012)Cambiaso (2013) on the Apache2 Aulds (2002) webserver. SlowLoris does not change necessarily the correct data flow path, but changes the rate in which certain operations repeat.
The Apache2 webserver is a multi-process program (we counted 6 processes as it loads, and 240 processes when attacked). Thus, we show here C-FLAT's multi-processes extension. Each httpd (this is the Apache2 Linux process name) instance may execute on any processor, and it is being invoked arbitrarily, therefore; we executed the C-FLAT attestation on all the processors as in Figure 7.
We attested the following apache2 routines:

```
unixd_accept
default_handler
core_create_req
core_create_conn
core_pre_connection
```

This resulted in 46 hooks, covering part of the program's flow of the webserver.
We compared the protected apache2 (version 2.2.3) to the not-protected in the Raspberry PI. We then used SlowHttpTest Slowhttp (2020) (version 1.6) to run the test as follows:

```
slowhttptest -c 500 -H -g -o ./outfile
-i 10 -r 200 -t GET -u http://192.168.1.13
-x 24 -p 2 -l 20
```

The above command means: (-c) initiate 500 connections, (-g) generate CSV files, (-H) use SlowLoris mode, i.e. send unfinished HTTP requests, in a rate (-r) of 200 connections per second, (-t) use the HTTP verb GET, (-x) follow up 24 bytes of data for SlowLoris and POST tests, and probe a connection for 2 seconds.
Detecting the attack
To detect the SlowLoris attack we recorded a legal access flow of the Firefox web browser (version 75.0 64bit), Chrome web browser (version 80.0.3987.132 64-bit) and Wget (version 1.19.4). We accessed the Apache2 remotely over wireless from an Ubuntu machine.
Table 7 demonstrates the program flow between an INIT and a QUOTE in the three runs. The hooked opcodes here are the pop operation (marked p) the branch operation (marked b). Pop is the hook for the "pop of frame pointer and returns address off stack" instruction, and "branch" is the hook for the "branch" instruction.

SlowLoris	b	b	b	b	p	b	b	p	b
Browser	b	b	b	b	b	b	b		
wget	b	b	b	b	b				

Table 7. Browser vs SlowLoris

From Table 7 it is evident that the number of operations is different, and the program flow is different. The INIT to QUOTE in the SlowLoris run belongs to a single process, and is a repeated snippet of 4600 operations recorded. Therefore, SlowLoris attack is detectable by C-FLAT.

Results
In Figures 11 and 10 we demonstrate the C-FLAT does not influence the flow of the webserver when we use SlowHttpTest. The bellow is one of the 5 runs of SlowHttpTest. There are no actual differences in the webserver with C-FLAT to the webserver without C-FLAT.

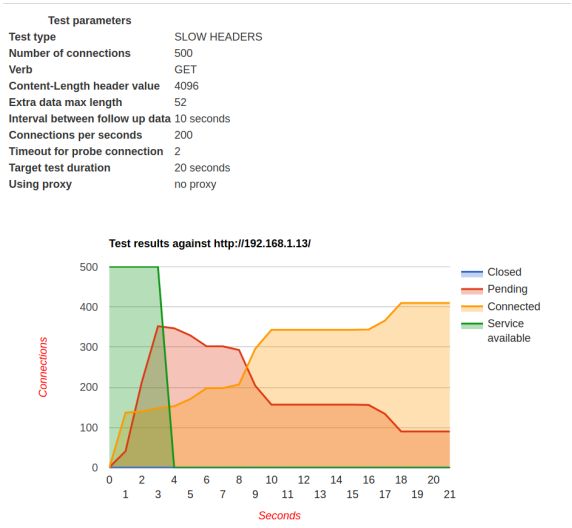


Figure 10. Slowloris Protected

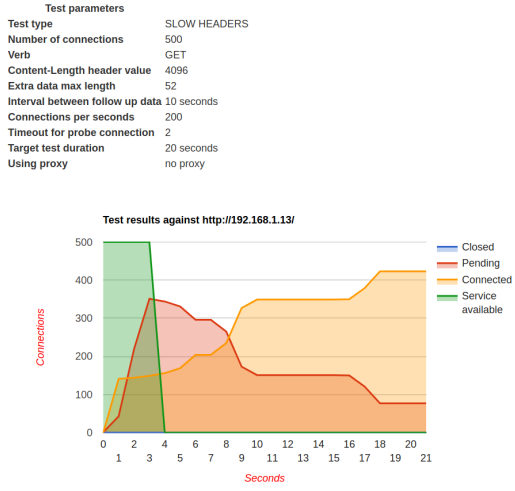


Figure 11. Slowloris Unprotected

The PI processor did not show any difference in its consumption between the two runs. The Apache2 consumed

on average, 2% processor cycles.

Conclusions

It is possible to protect Apache2 with C-FLAT. During the tests, C-FLAT demonstrated stability.

Summary

Future work

The trampoline is 40% of the total penalty. Therefore, to improve the mitigation, we intend to replace the trampoline code of the running process by the NOP opcode.

We intend to add to C-FLAT the ability to analyze and instrument a process dynamically. We intend to inject hooks into a running program. As the program loads, we examine in each context switch where the program counter is, record its position, and at some point, inject hooks to the most accessed code in the program.

Conclusions

It is only fair to say that other technologies, such as MOSKG Yan (2015) or KSP Liu (2018) provide only a few percentage overhead for micro-benchmarks. However, as noted, these technologies do not provide a versatile solution as C-FLAT Linux.

We conclude that though this paper lessons the performance penalty, there is still work to do in this area. Thus, C-FLAT Linux may be used in non-CPU intensive applications, such as web servers or command-line utilities, and it may be used also in an I/O intensive programs.

Glossary

VHE	Virtual Host Extention
EL	Exception Level
BKPT	Breakpoint
TEE	Trusted Execution Environment
RPC	Remote Procedure Call
ISR	Interrupt Service Routine
ELF	Executable and Linking Format
SMC	System Monitor Call
SVC	System Supervisor Call
Ver	Verifier
Prv	Prover
MOSKG	Multiple Operating Systems Kernel Guard
DKOM	Dynamic Kernel Object Manipulation
ROP	Return Oriented Programming
KSP	Kernel Stack Protect
DOS	Denial Of Service
CFI	Control Flow Inspection
CPI	Control Pointer Integrity
PAC	Pointer Authentication Code
NOP	No Operation
TLS	Thread Local Storage
QSEE	Qualcomm Secure Execution Environment
OP-TEE	Open Portable Trusted Execution Environment

Table 8. Abbreviations

References

Abera T, Asokan N, Davi L et al. C-flat: control-flow attestation for embedded systems software (2016). In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 743–754.

Yehuda RB and Zaidenberg NJ. Protection against reverse engineering in arm (2020). *International Journal of Information Security* 2020; 19(1): 39–51.

Flur S, Gray KE, Pulte C et al. Modelling the armv8 architecture, operationally: concurrency and isa (2016). In *ACM SIGPLAN Notices*, volume 51(1). ACM, pp. 608–621.

Tice C, Roeder T, Collingbourne P et al. Enforcing forward-edge control-flow integrity in gcc & llvm (2014). In *23rd USENIX Security Symposium (USENIX Security 14)*. pp. 941–955.

Abadi M, Budiu M, Erlingsson Ú et al. Control-flow integrity principles, implementations, and applications (2009). *ACM Transactions on Information and System Security (TISSEC)* 2009; 13(1): 4.

Dessouky G, Zeitouni S, Nyman T et al. Lo-fat: Low-overhead control flow attestation in hardware (2017). In *Proceedings of the 54th Annual Design Automation Conference 2017*. pp. 1–6.

Clements AA, Almakhdhub NS, Saab KS et al. Protecting bare-metal embedded systems with privilege overlays (2017). In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 289–303.

Moreira J, Rigo S, Polychronakis M et al. Drop the rop fine-grained control-flow integrity for the linux kernel (2017). *Black Hat*

Asia 2017; .

Davi L, Dmitrienko A, Egele M et al. Mocfi: A framework to mitigate control-flow attacks on smartphones. (2012). In *NDSS*, volume 26. pp. 27–40.

Van der Veen V, Andriesse D, Göktaş E et al. Practical context-sensitive cfi (2015). In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. pp. 927–940.

Pappas V. kbouncer: Efficient and transparent rop mitigation (2012). *Apr* 2012; 1: 1–2.

Seshadri A, Perrig A, Van Doorn L et al. Swatt: Software-based attestation for embedded devices (2004). In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE, pp. 272–282.

Seshadri A, Luk M, Shi E et al. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems (2005). In *Proceedings of the twentieth ACM symposium on Operating systems principles*. pp. 1–16.

Li Y, McCune JM and Perrig A. Viper: verifying the integrity of peripherals' firmware (2011). In *Proceedings of the 18th ACM conference on Computer and communications security*. pp. 3–16.

Eldefrawy K, Tsudik G, Francillon A et al. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. (2012). In *Ndss*, volume 12. pp. 1–15.

Koeberl P, Schulz S, Sadeghi AR et al. Trustlite: A security architecture for tiny embedded devices (2014). In *Proceedings of the Ninth European Conference on Computer Systems*. pp. 1–14.

Brasser F, El Mahjoub B, Sadeghi AR et al. Tytan: tiny trust anchor for tiny devices (2015). In *Proceedings of the 52nd Annual Design Automation Conference*. pp. 1–6.

Zhang C, Wei T, Chen Z et al. Practical control flow integrity and randomization for binary executables (2013). In *2013 IEEE Symposium on Security and Privacy*. IEEE, pp. 559–573.

Zhang M and Sekar R. Control flow integrity for cots binaries (2013). In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. pp. 337–352.

Kuznetsov V, Szekeres L, Payer M et al. Code-pointer integrity (2014). In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association. ISBN 978-1-931971-16-4, 2014. pp. 147–163. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

Evans I, Fingeret S, Gonzalez J et al. Missing the point (er): On the effectiveness of code pointer integrity (2015). In *2015 IEEE Symposium on Security and Privacy*. IEEE, pp. 781–796.

Nagarajan A, Varadharajan V, Hitchens M et al. Property based attestation and trusted computing: Analysis and challenges (2009). In *2009 Third International Conference on Network and System Security*. IEEE, pp. 278–285.

Sadeghi AR and Stüble C. Property-based attestation for computing platforms: caring about properties, not mechanisms (2004). In *Proceedings of the 2004 workshop on New security paradigms*. pp. 67–77.

Chen L, Landfermann R, Löhr H et al. A protocol for property-based attestation (2006). In *Proceedings of the first ACM workshop on Scalable trusted computing*. pp. 7–16.

Seshadri A, Luk M, Qu N et al. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses (2007). In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. pp. 335–350.

Liu W, Luo S, Liu Y et al. A kernel stack protection model against attacks from kernel execution units (2018). *Computers & Security* 2018; 72: 96–106.

Yan G, Luo S, Feng F et al. Moskg: countering kernel rootkits with a secure paging mechanism (2015). *Security and Communication Networks* 2015; 8(18): 3580–3591.

Azab AM, Ning P, Sezer EC et al. Hima: A hypervisor-based integrity measurement agent (2009). In *2009 Annual Computer Security Applications Conference*. IEEE, pp. 461–470.

Khen E, Zaidenberg NJ, Averbuch A et al. Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation (2013). In *2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. IEEE, pp. 78–85.

Zaidenberg NJ and Khen E. Detecting kernel vulnerabilities during the development phase (2015). In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. IEEE, pp. 224–230.

Kiperberg M, Leon R, Resh A et al. Hypervisor-assisted atomic memory acquisition in modern systems (2019). In *International Conference on Information Systems Security and Privacy*. SCITEPRESS Science And Technology Publications, pp. 155–162.

Ligh MH, Case A, Levy J et al. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory (2014)*. John Wiley & Sons, 2014.

Case A and Richard III GG. Memory forensics: The path forward (2017). *Digital Investigation* 2017; 20: 23–33.

Aljaedi A, Lindskog D, Zavarsky P et al. Comparative analysis of volatile memory forensics: live response vs. memory imaging (2011). In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, pp. 1253–1258.

Lu S, Lin Z and Zhang M. Kernel vulnerability analysis: A survey (2019). In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*. IEEE, pp. 549–554.

Lu H. Elf: From the programmer's perspective (1995), 1995.

Drepper U and Molnar I. The native posix thread library for linux (2003), 2003.

Ben Yehuda R and Zaidenberg N. Hyplets-multi exception level kernel towards linux rtos (2018). In *Proceedings of the 11th ACM International Systems and Storage Conference*. pp. 116–117.

Ben Yehuda R and Zaidenberg N. The hyplet - joining a program and a nanovisor for real-time and performance (2020). In *SPECTS, The 2020 International Symposium on Performance Evaluation of Computer and Telecommunication Systems Conference*. pp. 1–8.

Dall C and Nieh J. Kvm/arm: The design and implementation of the linux arm hypervisor (2014). In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 42. New York, NY, USA: Association for Computing Machinery, p. 333–348. DOI:10.1145/2654822.2541946. URL https://doi.org/10.1145/2654822.2541946.

Open portable trusted execution environment (2010). URL http://www.op-tee.org. Accessed: 2010-09-30.

Corbet J. Bpf comes to firewalls, 2018. URL https://lwn.net/Articles/747551/.

Wang Z and Jiang X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity (2010). In *2010 IEEE Symposium on Security and Privacy*. IEEE, pp. 380–395.

Damon E, Dale J, Laron E et al. Hands-on denial of service lab exercises using slowloris and rudy (2012). In *proceedings of the 2012 information security curriculum development conference*. pp. 21–29.

Cambiaso E, Papaleo G, Chiola G et al. Slow dos attacks: definition and categorisation (2013). *International Journal of Trust Management in Computing and Communications* 2013; 1(3-4): 300–319.

Aulds C. *Linux apache web server administration (2002)*. Sybex, 2002.

Slowhttptest linux man page (2020), 2020. URL https://linux.die.net/man/1/slowhttptest. Accessed: 2020-03-30.

PXI

ATTACKING TRUSTZONE

by

Ron Stajnord, Raz Ben Yehuda and Nezer Zaidenberg.

Submitted

Noname manuscript No.
(will be inserted by the editor)

Attacking TrustZone

Ron Stajnrod · Raz Ben Yehuda · Nezer Zaidenberg

Received: date / Accepted: date

Abstract ARM TrustZone offers Trusted Execution Environment (TEE) embedded into the processor cores. Some vendors offer ARM modules that do not fully comply with TrustZone specifications, which may lead to vulnerabilities in the system. In this paper, we present a DMA attack tutorial from the Insecure orld onto the Secure World and the design and implementation of this attack in a real insecure hardware.

Keywords TrustZone · Security

1 Introduction

The development of the Internet of Things (IoT) is hailed as the third wave of world information development after computers and the Internet [49], with embedded systems as the driving force for technological development in many domains, such as automotive, healthcare and industrial control in the emerging post-PC era. As an increasing number of computational and networked devices are integrated into all aspects of our lives in a pervasive and 'invisible' way, security becomes critical for the dependability of all smart or intelligent systems built upon these embedded systems [34].
Embedded IoT products are increasingly wireless. By their nature, such products are constrained in terms of computing and memory capacity and what can be done given cost realities [25]. The constrained nature of such devices means we are trying to 'build a fortress from pebbles', so to speak. Therefore, we must take the very best security measures to prevent malicious activity on those devices given the limited conditions, which often means cutting corners compared with other resource-rich areas of computing (personal computers, servers, etc.).
ARM TrustZone [2] was introduced as part of the ARMv6 architecture and is widely used in smartphones, tablets, wearables and other devices. As TrustZone is becoming a popular hardware security architecture for mobile devices and IoT, it is important to ensure the security of TrustZone itself [51].
Even though ARM TrustZone is a great way to implement security mechanisms across IoT-embedded devices, it is still prone to bad hardware and software implementations; thus, the hardware of different companies like Google, Samsung, Huawei, etc. might still be affected by severe vulnerabilities that compromise the entire security suite [38, 8, 28, 20].
Some ARM modules lack AMBA AXI [42] support, which leads to insecure memory separation between the Normal and Secure Worlds. In this paper, we present Direct Memory Access (DMA) attack [24] on ARM TrustZone Trusted Applications (TA) running in Open Portable Trusted Execution Environment (OP-TEE) [33, 17]. This allows an attacker to execute arbitrary code in the Secure World or read arbitrary data from the secure world into the rich OS. Our attack is a control-flow attack [12] [50] on the OP-TEE kernel.
In this paper, we show a hardware vulnerability on SoC that compromises ARM TrustZone. Using DMA attack, we gain the ability to replace trusted applications with

Ron Stajnrod
Interdisciplinary Center, Herzliya, Israel
E-mail: ronst12@gmail.com

Raz Ben Yehuda
University of Jyväskylä, Finland
E-mail: raziebe@gmail.com

Nezer Jacob Zaidenberg (correspondence author)
College of Management Academic Studies, Israel
University of Jyväskylä, Finland
E-mail: scipio@scipio.org

malicious ones. We demonstrate an attack on a Raspberry PI computer and explain how this method affects other platforms. This paper also provides measures to mitigate this vulnerability.

2 Background

2.1 ARM permission model

ARM has a unique approach to security and privilege levels. In ARMv7, ARM introduced the concept of secured and non-secured worlds through the implementation of TrustZone and starting from ARMv7a. ARM presents four exception (permission) levels as follows.

Exception Level 0 (EL0) Refers to the user-space code. Exception Level 0 is analogous to 'ring 3' on the x86 platform.
Exception Level 1 (EL1) Refers to the operating system code. Exception Level 1 is analogous to 'ring 0' on the x86 platform.
Exception Level 2 (EL2) Refers to HYP mode. Exception Level 2 is analogous to 'ring -1' or 'real mode' on the x86 platform.
Exception Level 3 (EL3) Refers to TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no directly analogous modes but related concepts in x86 are Intel's ME or SMM.

Each exception level provides its own set of special purpose registers and can access these registers at the lower levels, but not higher levels. The general purpose registers are shared; therefore, moving to a different exception level on the ARM architecture does not require the expensive context switch associated with the x86 architecture.

2.2 ARM TrustZone

ARM TrustZone technology is aimed at establishing trust in ARM-based platforms. In contrast to a TPM (Trusted Platform Module), which is designed as a fixed-function device with a predefined feature set, TrustZone represents a much more flexible approach by leveraging the CPU as a freely programmable trusted platform module. To do that, ARM introduced a special CPU mode called 'secure mode' in addition to the regular normal mode, thereby establishing the notions of a 'Secure World' and a 'Normal World' (Figure 1). The distinction between these worlds is completely orthogonal to the normal ring protection between user-level and kernel-level code, and hidden from the operating system running in the Normal World [3].

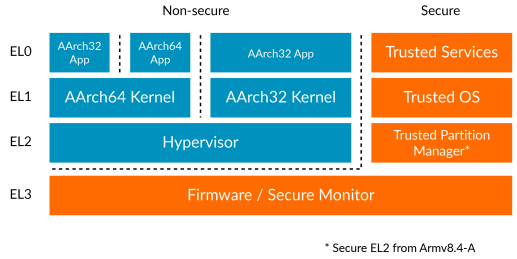


Fig. 1 Normal and Secured Worlds

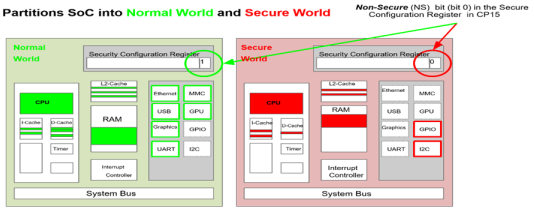


Fig. 2 NS Bit

As an example, the Linux kernel runs in EL1 and the user-space processes execute in EL0. The separation of Secure and Normal World secures certain RAM ranges and peripherals, which are only accessible by the Secure World. This means that a compromised Normal World code (in the user-space or the kernel) cannot access these memory ranges or devices. This separation is completely artificial. The same cores are used to run both Secure and Normal Worlds and they use the same RAM (Figure 2). The Non-Secure (NS) bit is used to determine whether the CPU executes in Normal or Secure World context to create a separation in memory. TrustZone technology extends beyond the processor into the SoC peripherals connected with the SoC, such as the DRAM controller (Figure 2), the DMA (Direct Memory Access), the secure boot ROM, the GIC (Generic Interrupt Controller), the TrustZone Address Space Controller (TZASC), the TrustZone Protection controller (TZPC) and the Dynamic Memory Controller (DMC). The above components communicate through the AXI bus and the SoC communicates with peripherals through the AXI_to_APB bridge. The SoC peripherals are implemented by third-party companies; therefore, to reduce costs, some vendors choose not to comply entirely with TrustZone specifications.
It is possible to access the entire memory from the Secure World but not vice versa. The Secure Monitor Call

(SMC) instruction is used to traverse to the Monitor in EL3. The SMC depends on the manufacture implementation and, thus, is prone to bugs and other vulnerabilities [20]. This paper focuses on the physical level of memory isolation.
TrustZone enables memory partitions between Normal and Secure Worlds by using the TZASC and the TZPC. This provides a secure I/O to peripherals over standard interfaces. For instance, the SPI or GPIO route interrupts to the TEE kernel (Secure World kernel) through the TZPC. The NS bit is used to secure on-chip peripherals from the Rich Execution Environment (REE, Normal World) [6]. TZASC utilises the NS bit for a memory-mapped device like DRAM. These two devices require support from the AXI bus, which is vendor-specific.
TrustZone use cases include building a root-of-trust for the system with everything needed for a secure boot and system recovery. Secure World trusted applications may be used for secure PIN and biometric checks to ensure details are safe from hacking. Another trusted application use case is Digital Right Management (DRM) for online media, where the private information is kept within the Secure World so hackers cannot access the keys required to reverse-engineer the system. Many more use cases of TrustZone can be found for IoT and mobile devices [31].

2.3 OP-TEE

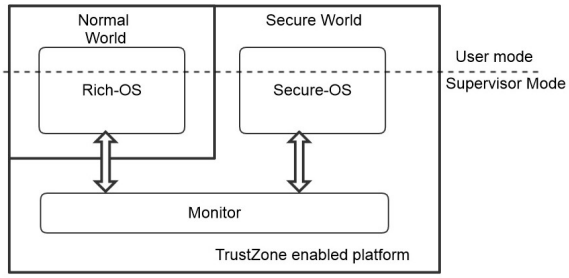


Fig. 3 Outline of Trust Zone

OP-TEE [33] is a Trusted Execution Environment (TEE) designed as a companion to a non-secure Linux kernel running on ARM Cortex-A cores using the TrustZone technology. OP-TEE implements TEE Internal Core API v1.1.x, which is the API exposed to Trusted Applications and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. These APIs are defined in the GlobalPlatform API specifications.

The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications.

OP-TEE is designed primarily to rely on the ARM TrustZone technology as the underlying hardware isolation mechanism. However, it has been structured to be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated processor core. The main design goals for OP-TEE are:

- **Isolation** - OP-TEE provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other by using underlying hardware support.
- **Small footprint** - OP-TEE should remain small enough to reside in a reasonable amount of on-chip memory as found on ARM-based systems.
- **Portability** - OP-TEE is aimed to be pluggable to different architectures and must support various setups such as multiple client OSs or multiple TEEs.

OP-TEE offers threads and shared memory among the REE to the secured OS, Secured interrupts, RPC from the secured to the REE and communication from the REE to the Secured World via the SMC interface where some are possible attack vectors. For instance, consider an attack on the SMC interface. It is possible to replace the SMC interface from the REE side with malicious code that hijacks the SMC requests in the non-secure side, for example, by manipulating the kernel code itself by a DMA attack. It is also possible to attack the shared-memory in cases when it is used.
TrustZone-protected DRAM or non-secure DRAM is used as the backing store. The data in the backing store are protected with a hash. However, read-only pages are not encrypted because the OP-TEE binary itself is not encrypted. Therefore, a DMA attack on the OP-TEE kernel is easier than on TA-encrypted programs as it bypasses the MMU permissions model as well as the need to encrypt the code.
Each TA is encrypted with a private key. The vendor creates a public key that is used to decrypt the TA. The decryption takes place in OP-TEE in the TrustZone. Thus, the program in its decrypted form is only visible in the Secured RAM and the processor's EL3 cache. It is, therefore, sensible to attack in the decryption area.

3 The DMA Attack

Direct Memory Access (DMA) allows I/O devices to access the memory. DMA has evolved since its inception, when a single DMA controller was set in a computing system. Following the introduction of many high-speed I/O peripherals, devices started to incorporate DMA

engines that enabled them to initiate DMA transactions without the coordination of a central DMA controller. ARM implements the advanced microcontroller bus architecture (AMBA), an open standard for on-chip interconnect specification. DMA transactions connect through the DMA controller to the on-SOC AMBA AXI Bus (AMBA advanced extensible interface) and the AMBA AXI Bus supports TrustZone NS-bit. The DMA controller can handle secure and non-secure events simultaneously, with full support for interrupts and peripherals. Examples of DMA devices are graphic cards, network adapters, FireWire, ThunderBolt, etc. Although DMA is essential for fast I/O transactions, it also opens new vulnerabilities to DMA attacks [24, 40, 5].

3.1 Attack Goal

The secured memory is accessible through DMA transactions. Through this vulnerability, the TrustZone can be exploited. We escalate privileges by reading data from the Secure World. Through this attack, we inject code to the Monitor in EL3, thus executing malicious programs in the Secure World operating system (the Secure World kernel). This offers us to bypass any validation of the secure operating system and also makes it possible to patch the EL1 kernel and execute arbitrary code.

3.2 The Attack - 'Trusted' Arbitrary Code Execution

Attack primitive is based on *Write What Where* vulnerability achieved using DMA transactions. We use this vulnerability to show that we can gain access to execute arbitrary code in the OP-TEE_OS, thereby bypassing OP-TEE OS trusted application signature validation and gaining control of every trusted application in the system. Our approach is to change the return values of key functions without changing the stack. This technique impedes CFI tools such as gcc stack guard [11] or Clang [29] kFCI to detect our attack.
Trusted applications are located on the REE file-system because this file-system usually contains more memory; by using this file-system, it is easier to update those applications. The trusted applications are built separately from the trusted operating system (similar to Linux kernel and user-space applications in the Normal World) and are signed with a private key from the manufacturer of the device application (e.g. Samsung sign their trusted applications with their private key). Common usages of trusted applications are DRM validations, HMAC (keyed-hash message authentication code) based one-time password, AES encryption and more. Using the trusted applications, the manufacturer of the device can make sure a compromised user or kernel will not break the integrity of the device. When the manufacturer wants to update a trusted application, they sign the new version with the same private key and distributes it to the users. When the Secure World OS executes a trusted application, if the signature is invalid, then a security error will occur and the program will not run. In our attack, we first use a DMA attack in order to read memory pages from the RAM. DMA attacks can be initiated from peripheral devices such as FireWire, PCI-connected devices (Network cards, GPU, etc.) as demonstrated by [43] [40] and [24]. DMA attacks can also be initiated from the CPU if the CPU can access the DMA controller. After reading memory pages from the RAM, we analyse the memory and compare it to ARM Trusted Firmware in order to locate similar functions. (Most of TrustZone software implementations are based on ARM Trusted Firmware, which makes reverse-engineering of the code simpler.) Moreover, there are some major vendors' secure OS (Trusted Execution Environment) in the market (QSEE, OPTEE) and we compare our memory dump to the compiled versions of those; by doing so, we can find the functions that validate trusted application signatures. Because none of the widely used TEE OS uses Address Space Layout Randomisation (ASLR) [10], we can use the address from our memory dump to override trusted applications signature validations with a DMA attack. After doing so, we can just replace any TA with our own malicious TA. Even though we do not know the correct signature private key, the TEE OS will succeed to validate our malicious TA.

4 Attack Evaluation

4.1 Raspberry PI Platform

We use a Raspberry PI3 Model B to demonstrate the attack. The Raspberry PI3 Model B's main specifications are shown in Table 1.

Soc	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2 GHz, (clocked to 700 MHz)
RAM	1GB LPDDR2 (900 MHz)
Clock	19.2 MHz

Table 1 PI3 specifications

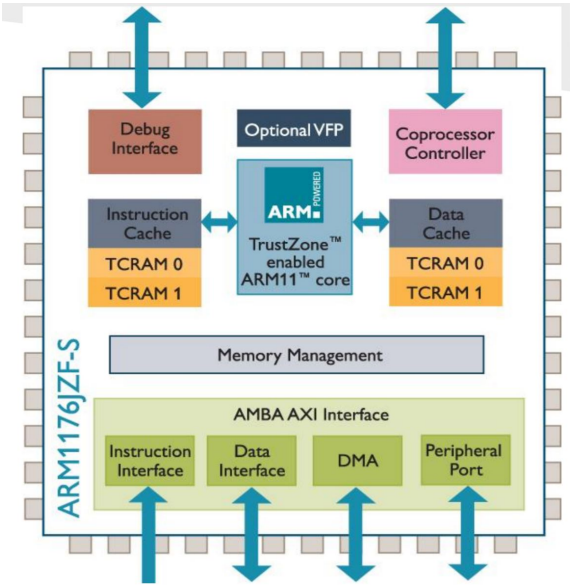


Fig. 4 BCM2387 overview

Figure 4 presents the BCM2837 chip. This Broadcom SOC supports TrustZone and DMA transactions through the AMBA *Advanced Microcontroller Bus Architecture* AXI (Advanced Extensible Interface). As mentioned earlier, not all TrustZone cores comply with the entire hardware specifications. Figure 4 shows that the BCM2837 has the correct AXI bus, but it lacks the TZASC and TZPC, making it vulnerable to DMA attacks.

4.2 OP-TEE for PI

OP-TEE supports *Raspberry PI 3 Model B*. The ARM Trusted Firmware is the basis for implementing Secure World software for the ARM A-Profile architectures (ARMv8-A and ARMv7-A), including an Exception Level 3 (EL3) Secure Monitor. The (SMC) Secure Monitor Call instruction is used to invoke functions between the Normal World and the Secure World through the Secure Monitor (Figure 3). ARM Trusted Firmware for the Raspberry PI provides a suitable starting point for the productization of Secure World boot and runtime firmware [1]. When a vendor uses OP-TEE on any hardware in general, and on Raspberry PI specifically, they will most likely use a trusted application in order to implement hardware security measures and secure their devices [30].
All real-world environment file-system trusted applications need to be signed. The signature is verified by OP-TEE OS upon loading of the TA. Within the OP-TEE OS source is a directory key. The public part of the key (public key) will be compiled into the OP-TEE OS binary and the signature of each TA will be verified against this key upon loading using an RSA signature scheme [37]. A vendor must sign his trusted application with a private key; thus, if a malicious party tries to change the trusted applications on the file-system, the OP-TEE OS will return a security error and will not execute the malicious trusted application. Without this mechanism, a malicious party would be able to alter trusted applications code, which will gain them access to the TrustZone security storage.
When a vendor updates a trusted application, they sign the new TA with their private key. OP-TEE OS contains the public key, thereby validating the TA. In this paper, we present a way to bypass this mechanism and execute our own 'trusted' applications.

4.3 Raspberry PI DMA

The DMA controller can be configured through the CPU as well as an external device. Therefore, we chose to perform this attack through the CPU. We authored a Linux kernel module to perform the DMA transactions. This module maps the DMA controller and configures the DMA control block to initiate DMA transactions. In OP-TEE's Linux kernel, the DMA controller address space is not available to the user-space. However, it is plausible to assume that an IoT device, for example, will enable this device for peripherals access. We argue an attack is possible in many IoT devices and we will show the following scenarios:

1. Some IoT devices mapping physical memory to the user-space to increase performance and save kernel access that may lead to DMA controller access.
2. Linux-based devices (IoT devices, routers, etc.) do not update their kernel versions very often due to compatibility issues and the large number of devices. Thus 'one day's' vulnerability can be used to exploit the device and gain root access to perform actions on the DMA controller [9], [44].
3. Attack peripheral device (Bluetooth/WIFI chip, SSD controller, etc.) to perform malicious DMA transactions [15], [46], [21].

All those scenarios may lead to a DMA attack and, on some devices, to a TrustZone vulnerability.

32-bit Word Offset	Description	Associated Read-Only Register
0	Transfer Information	TI
1	Source Address	SOURCE_AD
2	Destination Address	DEST_AD
3	Transfer Length	TXFR_LEN
4	2D Mode Stride	STRIDE
5	Next Control Block Address	NEXTCONBK
6-7	Reserved - set to zero.	N/A

Table 2 DMA Control Block Data Structure

32-bit Address Offset	Register Name	Description
0	CS	DMA Channel Control and Status
1	CONBLK_AD	DMA Channel Control Block Address
2	TI	DMA Channel Transfer Information
3	SOURCE_AD	DMA Channel Source Address
4	DEST_AD	DMA Channel Destination Address
5	TXFR_LEN	DMA Channel Transfer Length
6	STRIDE	DMA Channel 2D Stride
7	NEXTCONBK	DMA Channel Next CB Address
8	DEBUG	DMA Channel Debug

Table 3 DMA Controller

Table 2 presents the Control Block structure of a DMA in the Raspberry PI. Table 3 shows the DMA Controller registers. To initiate a DMA transaction, we first set the Control Block structure and then set *CONBLK_AD* in the DMA controller structure. We perform two types of DMA transactions:

1. Set *SOURCE_AD* to the Secure World physical address in order to read data of the Secure World.
2. Set *DEST_AD* to the Secure World physical address in order to write malicious code to the Secure World, thereby achieving arbitrary code execution.

5 The Attack - Evaluation on Raspberry PI

In OP-TEE environment. Trusted applications are signed with the key from the build of the original OP-TEE core blob. Trusted applications consist of a signed *ELF* header, named from the UUID of the trusted application (set during compilation time) and the suffix *.ta*. When a trusted application is replaced in the REE file-system with the new one, the signatures and UUID are validated by the OP-TEE OS (Figure 5).

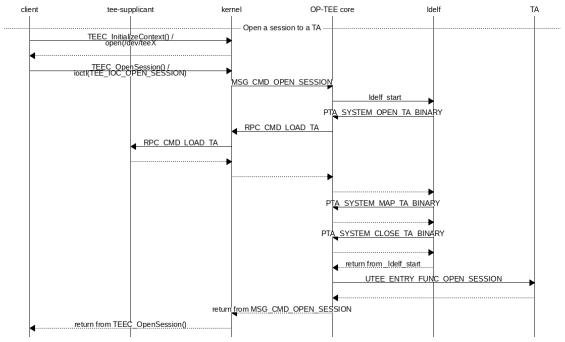


Fig. 5 Open session flow

Invoking a trusted application function from the Normal World requires the use of SMC (Secure Monitor Call). SMC is used to communicate between the Normal World and Secure World. SMC is initiated by the kernel (EL1) to reach the EL3 monitor. OP-TEE provides a Linux kernel driver to interact with the OP-TEE in TrustZone. For instance, *PTA_SYSTEM_OPEN_TA_BINARY* function is accessed by this driver to the OP-TEE OS in the Secure World. *PTA_SYSTEM_OPEN_TA_BINARY* calls *system_open_ta_binary*, which looks for the user-trusted application ELF by the UUID in the storage (file-system). After finding the trusted application ELF in the REE file-system, the OP-TEE OS loads the ELF header and maps the trusted application sections into the secure memory using *PTA_SYSTEM_MAP_TA_BINARY*. After loading the trusted application, the user is able to invoke the trusted application functionality through the OP-TEE Linux kernel driver.
We focus on two functions:
ree_fs_ta_open and *ree_fs_ta_read* called by *PTA_SYSTEM_OPEN* and *PTA_SYSTEM_MAP_TA_BINARY* respectively.
Trusted applications binaries contain a signed header so that a malicious user cannot replace the trusted applications. If a malicious user replaces a trusted application, then OP-TEE OS returns a security error when it tries to execute those trusted applications. In order for OP-TEE OS to validate those signatures, as a trusted application executes, the function *ree_fs_ta_open* loads the trusted application header, validates the application header signature (Figure 6) and validates its size (Figure 7). When OP-TEE OS maps the TA into the secure memory, it loads the application to the memory using *ree_fs_ta_read*, which validates the encrypted trusted application signature (Figures 8 and 9).

```
/* Validate header signature */
res = shdr_verify_signature(shdr);
if (res != TEE_SUCCESS)
  goto error_free_payload;

```

Fig. 6 ree_fs_ta_open Header signature validation

```
if (ta_size != offs + shdr->img_size) {
  res = TEE_ERROR_SECURITY;
  goto error_free_hash;
}

```

Fig. 7 ree_fs_open TA size validation

```
  if (handle->shdr->img_type ==
 SHDR_ENCRYPTED_TA) {
    /*
     * Last read: time to finalise
 authenticated
     * decryption.
     */
    res = tee_ta_decrypt_final(handle->
 enc_ctx,handle->ehdr, NULL, NULL, 0);
    if (res != TEE_SUCCESS)
      return TEE_ERROR_SECURITY;
  }

```

Fig. 8 ree_fs_ta_read decrypts a TA header

```
  /*
   * Last read: time to check if our
 digest matches the expected
   * one (from the signed header)
   */
  res = check_digest(handle);
  if (res != TEE_SUCCESS)
    return res;

```

Fig. 9 ree_fs_read validates the encrypted header against the hash of the plain header

In the first step, we reverse-engineered OP-TEE OS (using radare2 [36]) in order to find *key* opcodes of both functions to exploit (Figures 6 - 9). We used DMA transactions to read chunks of physical RAM in order to find the opcodes that match the functions above. Once we located the opcodes in the memory and noticed that these functions load in the same location in physical memory every time. We used DMA transactions to override the return values of the validations mentioned above (Figure 6 - 9), thereby gaining the ability to compile our own trusted application, sign it with an arbitrary key and execute it on the machine. We replaced two types of opcodes: the comparison opcode of *w0* register was replaced with *cmp w0,w0* so it always returned true and, when moving the return value of the function to *w0* register, we replaced this command with *eor w0,w0,0* so the value of *w0* register would be 0, again having the return values of the validation functions equal true. We were able to perform this replacement using just a simple DMA transaction with the control block *DEST_AD*, which contains the physical address of the opcodes we found, all of which constitute in the Secure World memory.
Because Raspberry PI does not support physical bus Secure World separation, we used a DMA transaction to replace the correct opcodes with our malicious opcodes. The trusted applications binaries are on the REE file-system and, thus, can be over-written; however, the OP-TEE OS will never execute a replaced TA when the signature does not match. Faking a matching signature requires finding a private key of *2048-bit* that matches the public key; therefore, only the owner of the key would be able to replace those applications.
In our case, we compiled a new TA with the same UUID of the original one and put it in the file-system location. By executing our malicious TA, we gained the ability to manipulate ARM TrustZone to execute invalid signed binaries. For example we compiled a fake AES TA (given in the examples of the OP-TEE suite) that encrypts data with our malicious key. Thus, every time the user uses this TA to perform AES, the data will not be truly encrypted with a secret key.

5.1 Other Attack Possibilities

Using the above method, it is possible to bypass the TA validation by replacing the signature key itself or by patching the validation function so there is no need to sign the TA at all. Using DMA attacks on the TrustZone gives a wide range of attack possibilities. In this paper, we show the usage of DMA attack to perform ACE (Arbitrary Code Execution); however, it is also possible to use this method to read arbitrary code from physical memory whereby a malicious user can access sensitive data.

6 Mitigation

When choosing an SoC for a device, you must compare the device requirements to the features the SoC contains. In our case, when choosing a SoC we want to make sure the SoC architecture has all the chips required for ARM TrustZone to work properly (TZASC,

TZPC, supported bus, etc.). The process of checking the SoC architecture is not always easy and surely not automatic because not all vendors publish their SoC architecture. We suggest for SoC vendors to be more transparent about their architecture when it comes to security features. We also suggest that manufacturers ensure their SoC hardware supports not only TrustZone ARM Core but also TrustZone specifications. In cases where a fully compatible TrustZone is not available (lack of hardware on the SoC that makes the TrustZone secure), we list other protection techniques:

- Using SMMU (similar to IOMMU on Intel x86) to configure specific addresses for DMA controllers. SMMU works as MMU for BUS access, so any memory access through the BUS would have to be matched to the permission configured to the accessed address. With SMMU and a correct configuration, a DMA attack through peripherals will not be possible. It is also important to note a kernel attacker could change this configuration.
- In the case of Raspberry PI, by disabling the DMA controller, a non-privileged user or peripheral would not be able to use DMA transactions.
- Set the Secure World on a different RAM without DMA controller mapping so there is no physical interface between the Normal and Secure Worlds.

A software technique would be to encrypt parts of the OP-TEE code itself, mainly the TA decipher functions. Only when these functions are used will OP-TEE decrypt the functions into the cache, validate the TA and evict the cache. This method was introduced by [48]. Using this method, an attacker would have to time his attack in order to get the code from the RAM. Combine this method with ASLR and HALT the other cores and this attack will be mitigated.

7 Related Work

Many words were written on side-channel attacks and other vulnerable targets in ARM architecture in prior research. In the area of ARM, [8] et al. describe a downgrade or rollback attack. A trusted application is encrypted for security purposes by public and private keys that originate from the hardware. In cases when the system is updated, old TAs can still be executed on the new system. A downgrade attack is when an attacker exploits a vulnerability in the old TA version by patching the old version onto the new TA version. According to [8], the above applies to the OP-TEE and QSEE (Qualcomm's Secure Execution Environment). [8] et al. describe a simple procedure for mobile phones: root the device, remount the 'system' partition in READ-WRITE mode, replace the current trustlet with an old vulnerable trustlet and use the trustlet. [8] et al. describe another possible rollback attack on the chain of trust and proves it possible to downgrade the bootloader successfully.
Armageddon [27] et al. explore attacks on ARM caches, concentrating on cross-core cache attacks in non-rooted arm mobile devices and showing a novel approach to exploit the coherence protocols. Although most smartphones have multiple processors that do not share caches, cache coherence protocols allow processors to fetch cache lines. By exploiting the lack of $cacheflush$ on 'old' ARM cores (before ARMv8), a novel technique that analyses cache eviction strategies and another approach on how to perform cycle-timing without root access. Armageddon [27] et al. provide a technique to gain sensitive information such as inter-keystroke timings or the length of a swipe action requiring significantly higher measurement accuracy. As for TrustZone vulnerability, Armageddon [27] shows a cache attack used to monitor cache activity caused within the ARM TrustZone from the Normal World. Flush and Reload attack [47] et al. take advantage of the coherence protocol in a multiprocessor computer. In most ARM processors, the last level cache is inclusive (i.e. it includes low-level cache lines); therefore, examining the content of the last-level cache may provide the contents of low-level cache lines of another core. However, the AutoLock [18] tool assesses the real risk in cache attacks, prevents cross-cache evictions, and highlights the intricacies of cache attacks in ARM. [18] et al. claim that unlike Intel processors, many ARM caches are both inclusive and exclusive, and therefore hardens the LLC (last-level cache) attacks. In their work, Demme et al. [14] demonstrate that small changes to the cache architecture have a considerable impact on side-channel vulnerability.
Like cache attacks, DMA attacks are continuously under research. [43] et al. show that by dumping memory frequently enough using DMA transactions, write patterns can be examined, and some algorithms, such as the RSA Montgomery ladder [22], may leak secrets. DAGGER [40], a DMA-based keystroke logger, exfiltrates captured data to an external entity and cannot be detected by anti-virus. [40] shows how DAGGER can steal cryptographic keys, target OS kernel structure, and copy files from the file cache on Linux and Windows through DMA malware, even if the memory addresses are random. [40] et al. also offer countermeasures to detect DMA attacks. [7] et al. integrate DMA attacks through FireWire into Metasploit [23]. Thus, an attacker could use Metasploit [23] for payload selection, session control, etc. and attack via DMA over

Firewire. TRESOR-HUNT [5] relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory but can write arbitrary values to memory as well. Hard disk encryption keys were considered safe if not saved on the RAM, but TRESOR-HUNT [5] injects malicious code to the kernel using DMA attack and then extracts disk encryption keys from the CPU into the target system's memory from which they can be retrieved using a normal DMA transfer. [41] et al show that an adversary with physical access to a device, could impersonate the device's memory controller, by attaching a malicious memory controller to the exposed pins of each DIMM socket of RAM and, by doing so, an attacker would have full access (READ/WRITE) to the target memory. Duflot et al. [15] introduce the vulnerability of remote code execution on a network adapter and how it could compromise the system-running kernel using DMA attack. BROADPWN [4] is a novel approach of privilege escalation. From exploiting a bug in Broadcom WiFi chip into DMA attack on the main processor of the device.
The emerging of cache, DMA, and hardware attacks shows that not only software bugs can impose security risks but also hardware implementation bugs are becoming more common, specifically when new features rely on old security assumptions. [32] et al.show that because ARMv7 (the ARM debugging model) requires no physical access, a low-privilege host can use ARM debugging features to gain read/write access to TrustZone Secure World. Because there is no hardware privilege access control, a low-privilege host can initiate a debug session with a high-privilege target using ARM debugging features. [32] et al.use ARM debugging features to leak private keys from the Secure World, thus compromising ARM TrustZone security. The hardware implementation bugs of ARM debugging features affect development boards, IoT devices, and mobile devices. Defense against these vulnerabilities requires hardware and software solutions like the vulnerability we found. [32] et al.suggest that ARM should add restrictions in the interprocessor debugging model to enforce permission between host and target. SoC vendors should refine debug signal management, and add support to disable only inter-processor debugging. OEMs should add software-based access control to go with the hardware permission model. Matt Spisak et al.[39] describe another processor feature-based attack using ARM CoreSight debug features. [39] et al.leverage ARM PMU (Performance Monitoring Unit) to create a rootkit that cannot be detected by the kernel monitor because it does not change the kernel syscall but rather attaches through the PMU to any syscall. Thus, every syscall will raise a PMU event, and the rootkit would be able to modify the input and output data of the syscall. This attack is possible due to a hardware implementation bug of a debug signal authorization that enables debug features in the hardware. [48] et al.suggest a different approach where the code and data that need to be protected are kept only in EL2 [16] (HYP mode) instead of in the TrustZone, where there is a strong coupling between vendor-specific code and hardware implementation; in which case, EL1 and EL0 will not have access to this code. Cloaker [13] et al.leverage ARM architecture System Control Register (SCTLR) to move the exception vector table (EVT) from high to low address so that mapping a malicious EVT at address 0x0 would intercept all exceptions.
Much is found in the literature on control-flow integrity (CFI). [29] et al.present the kernel CFI used to protect the kernel's stack and heap. A flaw in the kernel may allow user processes to write to kernel-space. Therefore, processor vendors presented the NX (Never Execute) bit that thwarts execution from the kernel's data portions. However, the execution segments were still writable and vulnerable to exploits. This led to making the kernel execution part read-only. But this also was not enough, as all of the user-space portion could be both written to and executed via a kernel exploit. To probity this, Intel created the supervisor mode execution prevention (SMEP) and ARM privileged execute never (PXN) bit. These features restrict the kernel from executing user-space memory while in kernel mode. This led attackers to target the stack, mainly manipulating the return addresses kept on the stack. This type of attack is referred to as 'return-oriented programming' (ROP) attacks. ROP attacks manipulate indirect calls, i.e. function pointers. These attacks concentrate on the calling (forward edge) and returning (backward edge) of a function. Thus, the main purpose of CFI is to try to ensure that forward edges go to the expected addresses and that the backward edges are not changed. CFI is implemented through the Clang compiler extensions and utilizes link-time optimization (LTO) to examine the entire kernel code. Functions are classified according to their signature and checked in runtime. Another mechanism is kCFI, which narrows the classification of the edges. Thus, to use this feature, OP-TEE must be compiled with Clang and then apply kFCI on it. Unfortunately, none of these defenses thwart a DMA attack.
In the area of thwarting hypervisor CFI attacks, [45] et al. offer Hypersafe. Hypersafe is used to protect the hypervisor from CF hijack attack through a memory lockdown and restricts pointer indexing, a layer of indirection that converts the control data into pointer indexes. These pointer indexes are restricted such that

the corresponding call/return targets strictly follow the hypervisor control flow graph, hence expanding protection to control-flow integrity.
This mitigation reduces the ease of performing a DMA attack on the hypervisor and, combined with IOMMUs, DMA attacks can be entirely mitigated.

8 Conclusions

Using the above method, it is possible to bypass the TA validation by replacing the signature key itself or patching the validation function so there is no need to sign the TA at all. Using DMA attacks on the TrustZone gives a wide range of attack possibilities. In this paper, we show the usage of DMA Attack to perform ACE (Arbitrary Code Execution). However, it is also possible to use this method to read arbitrary code from physical memory whereby a malicious user can access sensitive data. We also show that hardware implementation bugs are common even on security features like ARM TrustZone.

Manufacturer	SoC	Device
Texas Instrument	CC2538	Sensibo
HISILICON	Hi3518EV200	Security Cameras
HISILICON	Hi3519V101	Security Cameras
Amlogic	Meson3	Wifi Network Speaker
ST	STM32	Smart Vacuums

Table 4 List of vulnerable SoCs

Table 4 presents a few SoCs of real IoT devices that lack some TrustZone hardware, but support TrustZone in the ARM Core, thus making the TrustZone 'untrusted'. One can claim the devices using this SoC do not use TrustZone at all; However, if this were the case, then those devices would not be using all the security options given to them, thus introducing architecture security flaws [26], [19], [35].

9 Compliance with Ethical Standards

9.1 Disclosure of potential conflicts of interest

Not applicable.

9.2 Research involving human participants and/or animals

Not applicable.

9.3 Informed consent

Not applicable.

References

1. Arm trusted firmware. `https://github.com/ARM-software/arm-trusted-firmware`. Accessed: 2020-04-15.
2. Arm trustzone. `https://developer.arm.com/ip-products/security-ip/trustzone`. Accessed: 2020-04-15.
3. A technical report on tee and arm trustzone. `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone`. Accessed: 2020-04-16.
4. Nitay Artenstein. Broadpwn: Remotely compromising android and ios via a bug in broadcom's wi-fi chipsets. *Black Hat USA*, 2017.
5. Erik-Oliver Blass and William Robertson. Tresorhunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78, 2012.
6. O. Blazy and C.Y. Yeun. *Information Security Theory and Practice: 12th IFIP WG 11.2 International Conference, WISTP 2018, Brussels, Belgium, December 10–11, 2018, Revised Selected Papers*. Lecture Notes in Computer Science. Springer International Publishing, 2019.
7. Rory Breuk and Albert Spruyt. Integrating dma attacks in metasploit. In *Sebug: http://sebug. net/paper/Meeting-Documents/hitbsecconf2012ams D*, volume 2, 2012.
8. Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.

9. Nikolai Hampton (Computerworld. The working dead: The security risks of outdated linux kernels. https://www2.computerworld.com.au/article/615338/working-dead-security-risk-dated-linux-kernels/. Accessed: 2020-04-16.
10. Kees Cook. Kernel address space layout randomization. *Linux Security Summit*, 2013.
11. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.
12. Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.
13. Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. Cloaker: Hardware supported rootkit concealment. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 296–310. IEEE, 2008.
14. John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 106–117. IEEE, 2012.
15. Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.
16. Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: concurrency and isa. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 608–621, 2016.
17. Christian Göttel, Pascal Felber, and Valerio Schiavoni. Developing secure services for iot with optee: a first look at performance and usability. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 170–178. Springer, 2019.
18. Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on {ARM} are harder than you think. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1075–1091, 2017.
19. Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501, 2017.
20. Joffrey Guilbon. Attacking the arm's trustzone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html. Accessed: 2020-04-16.
21. The latest security information on intel® products. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00266.html. Accessed: 2020-04-15.
22. Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International workshop on cryptographic hardware and embedded systems*, pages 291–302. Springer, 2002.
23. David Kennedy, Jim O'gorman, Devon Kearns, and Mati Aharoni. *Metasploit: the penetration tester's guide*. No Starch Press, 2011.
24. Gil Kupfer, Dan Tsafrir, and Nadav Amit. *IOMMU-resistant DMA attacks*. PhD thesis, Computer Science Department, Technion, 2018.
25. John Leonard. Why trustzone matters for iot. https://blog.nordicsemi.com/getconnected/why-trustzone-matters-in-iot. Accessed: 2020-04-15.
26. Christian Lesjak, Daniel Hein, and Johannes Winter. Hardware-security technologies for industrial iot: Trustzone and security controller. In *IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society*, pages 002589–002595. IEEE, 2015.
27. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 549–564, 2016.
28. Slava Makkaveev. The road to qualcomm trustzone apps fuzzing. https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/. Accessed: 2020-04-16.
29. João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. Drop the rop fine-grained control-flow integrity for the linux kernel. *Black Hat Asia*, 2017.
30. A. Nehal and P. Ahlawat. Securing iot applications with op-tee from hardware level os. In *2019 3rd International conference on Electronics, Communi-*

cation and Aerospace Technology (ICECA), pages 1441–1444, 2019.
31. B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.
32. Zhenyu Ning and Fengwei Zhang. Understanding the security of arm debugging features. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 602–619. IEEE, 2019.
33. Open portable trusted execution environment. `http://www.op-tee.org/`. Accessed: 2010-09-30.
34. Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152. IEEE, 2015.
35. Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. Iioteed: an enhanced, trusted execution environment for industrial iot edge devices. *IEEE Internet Computing*, 21(1):40–47, 2017.
36. Libre and portable reverse engineering framework. `https://rada.re/n/`. Accessed: 2020-04-17.
37. Ronald L Rivest, Adi Shamir, and Leonard M Adleman. Cryptographic communications system and method, September 20 1983. US Patent 4,405,829.
38. Di Shen. Exploiting trustzone on android. *Black Hat USA*, 2015.
39. Matt Spisak. Hardware-assisted rootkits: Abusing performance counters on the {ARM} and x86 architectures. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.
40. Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.
41. Anna Trikalinou and Dan Lake. Taking dma attacks to the next level. *BlackHat USA*, 2017.
42. About the axi trustzone memory adapter. `https://developer.arm.com/docs/dto0017/a/about-the-axi-trustzone-memory-adapter`. Accessed: 2020-04-15.
43. Marten van Dijk, Syed Kamran Haider, Chenglu Jin, and Phuong Ha Nguyen. Advanced power side channel cache side channel attacks dma attacks, 2017.
44. Jake Wallen. Most iot devices are an attack waiting to happen, unless manufacturers update their kernels. `https://www.techrepublic.com/article/most-iot-devices-are-an-attack-waiting-to-happen-unless-manufacturers-update-their-kernels/`. Accessed: 2020-04-16.
45. Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, 2010.
46. Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *WOOT*, pages 12–21, 2012.
47. Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.
48. Raz Ben Yehuda and Nezer Jacob Zaidenberg. Protection against reverse engineering in arm. *International Journal of Information Security*, 19(1):39–51, 2020.
49. Meiyu Zhang, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. Softme: A software-based memory protection approach for tee system to resist physical attacks. *Security and Communication Networks*, 2019, 2019.
50. Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.
51. Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Minimal kernel: An operating system architecture for {TEE} to resist board level physical attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 105–120, 2019.

6 ERRATA

The errata is divided into essential corrections in the dissertation and corrections in the papers.

6.1 Essential Corrections in the Dissertations

6.1.1 C1

Here are the links to the code:
Paper PI: https://github.com/raziebe/rasp-hyplet branch=isolets
Paper PII: https://github.com/raziebe/rasp-hyplet branch=hyplet
Paper PV: https://github.com/raziebe/TEE branch=rt_mappings2
Paper PVI: https://github.com/raziebe/rasp-hyplet branch=hyplet
Paper PVII: https://github.com/raziebe/rasp-hyplet branch=honeypot
Paper PVIII: https://github.com/raziebe/LiME branch=arm64
Paper PVIII: https://github.com/raziebe/volatility branch=arm64
Paper PVIII: https://github.com/raziebe/rasp-hyplet branch=mem_acq
Paper PIX: https://github.com/raziebe/rasp-hyplet branches=offlet,ultrasonic
Paper PX: https://github.com/raziebe/rasp-hyplet branch=cflat
Paper PXI: https://github.com/ronst22/dma_repo branch=master

6.1.2 C2

Page 26. Section 2.3.2. Cybersecurity challenges. Redo sentence "Malware, sometimes ..."

6.1.3 C3

Page 26: Section 2.3.2.1 Malware challenges. Redo sentence "Usually".

6.1.4 C4

Page 26: Section 2.3.2.1 Malware challenges. Redo sentence "Man in Middlle". Remove sentence "Social Engineering".

6.1.5 C5

Page 27: Section 2.3.2.2. "Computer Forensics Challenges". Remove sentence "While the operating system runs".

6.1.6 C6

Page 27: "LiME". Bottom of page: "Lime is a driver..."

6.1.7 C7

Page 38: "Thin Hypervisor". Added References to the sentence starts "A thin hypervisor".

6.1.8 C8

Page 44: section 3.3. "Security of thin hypervisors". Removed line, "In general..". Remove line "The open-source".

6.1.9 C9

Page 47: In section 4.1.2. Remove AES and elliptic curve. Leave only "we make no assumption about the encryption".

6.1.10 C10

Page 53: In section 4.3.3 , rephrase "We record" sentence.

6.1.11 C11

Page 56: Under Figure 18. Redo the sentence: "From Figure 18 we can see that".

6.1.12 C12

Page 58: Threat Model section. Redo starting sentence "An attacker..".

6.1.13 C13

Page 61: Section 4.6.3. Hyperwall technique. add to the sentence that starts "Only packets", the words "through sendto, sendmsg and sendmmsg are considered legitimate".

6.1.14 C14

Page 62: Hyperwall: Section 4.6.5 Future work. Redo The sentence that starts "If a CFI based filtering...".

6.1.15 C15

Page 65: Section 4.8.3. Remove the sentence "The vendor creates...".

6.2 Paper PI: The offline scheduler for embedded transportation systems

6.2.1 C16

In section III, page 3 in the paper, the formula (J.Flynn. (1995)) is incorrect. It should be:

$$O_p^T = \sum_i^p (O_i + o_i) \geq O_1$$

and :

$$o_{i+1} \geq o_i$$

6.2.2 C17

In section IV, Moore's law as stated is not correct. Moore law states Waldrop (2016) that the number of transistors per microprocessor has doubled about every two years.

6.2.3 C18

The opening of section VI, the OFFLINE TIMER, appears to be confusing. Here we meant that the offline timer is useful when it executes in high frequencies. Otherwise, it is better to use a regular interrupt timer.

6.2.4 C19

In section VI, the OFFLINE TIMER, it is said that when the amount of timers is larger, accuracy is less important. This phrase is incorrect. We meant that a large number of timed routines might cause delays.

6.2.5 C20

In section IX, conclusion. The sentence "Who does need network processors for..". This phrase is incorrect. The Offline scheduler consumes processors from the host machine, requires Linux, etcetera.

6.3 Paper PII: Hyplets - Multi Exception Level Kernel towards Linux RTOS - Systor

6.3.1 C21

The interrupt latency mentioned in section 3.1 is not backed up with benchmarks.

The first benchmark we used did not produce enough data. Therefore, we repeated the test in the extended paper of Hyplet (paper PVI). The results were an average of 3.9 μs, a maximum of 9μs, and the minimum was 1.7μs.

6.3.2 C22

In the table, Normal Linux supersedes RT_PREEMPT in some measures. The reason is that the RT_PREEMPT tends to fault in Raspberry Pi3 in kernel version 4.4.x. Therefore, we conducted a more robust benchmark in the Hyplet paper (PVI), section C, Table III, and the results were more sensible. The difference is because we used a different kernel version 4.19 for the RT_PREEMPT.

6.4 Paper PIII: ARM Security Alternatives

6.4.1 C23

We detail Kinibi, so it is not left out the survey. On page 604, we say we mainly consider open source alternatives. However, we felt it is still essential to list QSEE and Kinibi as they are widespread operating systems.

6.5 Paper PIV: Hyperwall

6.5.1 C24

Page 2. The attack module is artificial because we assume in the paper zero-day. We also note that this attack may be employed on other operating systems, such as Android, where a vulnerability might be exploited to perform this sort of attack.

6.5.2 C25

Page 7. section 5.3. We did not cover write, writev over sockets, splice, sendfile and possibly other system calls. This is correct and requires further research and would have to be covered for a realistic system evaluation. The write syscall, for

example, passes the file descriptor, the buffer and its size to the kernel. We believe that it is possible to propagate "write" arguments to HYP mode.

6.5.3 C26

Page 8. ICMP. We did not cover the ICMP case.

6.5.4 C27

Page 9: Performance is degraded due to buffer access and searching in the balanced tree. A performance penalty is expected when applying security. We, therefore, offer to use this technique on low-performance devices that generate less network traffic. We also continue this work and try to apply CFI as noted in future work.

6.5.5 C28

Page 11: Phoronix LM bench. Phoronix benchmark is a reliable test suite that tests various aspects of the target device. It tests the processor, the RAM, the GPU, the storage, the network, etcetera. We believe it provides an overall view of the system under stress. In the benchmark on page 12, we can see the total overhead of our technology.

6.6 Paper PV: Protection against reverse engineering in ARM

6.6.1 C29

Page 5, section 2.5 claims: "Once a trustworthy hypervisor is running". However, this is not correct; an example, the Spectre vulnerability can be exploited to manipulate the hypervisor Kocher et al. (2019).

6.6.2 C30

In section 5.1, Program protection in ARM, it is a wrong claim that the elliptic curve is an encryption function and that AES was never broken. We removed the sentence. Our TEE technology is indifferent to the encryption and decryption algorithm.

6.6.3 C31

Page 6: Section 4. Anti Reverse Engineering. The footprint is 100KB because the decryption code is big. We offer to map the encryption code in real-time to the microvisor without the keys. The keys are expected to be put by EL3 software.

6.6.4 C32

Page 12, Debian version is hikey-rootfs-debian-jessie-alip-20160629-120.

6.6.5 C33

Page 12. Section 5.4. OS dependence. Refrain from accessing its own page tables. The hypervisor page tables are not mapped to the hypervisor, and it cannot modify them accidentally.

6.7 Paper PVI: The hyplet-Joining a Program and a Nanovisor for real-time and Performance - SPECTS 2020, IEEE

6.7.1 C34

In the Safety section, in the case of an endless loop that hogs the processor. It is doubtful to stop the loop via a guard (without a memory barrier) because the hyplet executes code from the cache, and therefore, the guard value does not propagate to the hyplet.
We, therefore, suggest using an NMI watchdog to execute in the hypervisor. Linux implements an NMI watchdog by setting the F bit in the DAIF register documentation arm (2021b). We offer to apply this technique in HYP mode as well.

6.7.2 C35

In section Protection against reverse engineering. The table in the paper does not reflect the expected overhead of the constant copying or decryption. Also, it is not clear why the overhead is too small.
In table 5 we add the overhead.

Iterations	Encrypted	Clear	overhead %
1	1185	1127	5
10	2737	2597	5
100	18022	18018	0
1000	173925	171251	1
10000	1758997	1670811	5

TABLE 5 Duration of stack access in ticks

The low overhead is since the stack used in this benchmark is small (16 bytes). The stack access does not cause a cache eviction. Therefore, the penalty of the stack copy or stack decryption is minor than expected. In paper PV, Table 8, we show ten pages stack access, and the penalty associated.

6.7.3 C36

Section B, The hyplet security & Privilege escalation in RTOS. The line "Many real-time implementations move userspace code from EL0 to EL1". This assumption is plausible. For example, please see the EtherLAB (https://www.etherlab.org/en/index.php) driver. The entire EtherCAT stack is implemented as a kernel driver.

6.7.4 C37

Section C: Static analysis to eliminate security concerns.
We coded a static analyzer that analyzes the code offline. The analyzer scans the hyplet section in the ELF binary for privileged commands. Commands such as "MSR", "MRS", "SVC", "HVC", or "SMC" are not allowed. The analyzer also searches for special registers, for example, TTBR_EL1 and TTBR_EL2. The static analyzer does not handle self-modifying code, and it does not handle jumps to computed targets. These targets are not part of the hyplet and are not executed in EL2. non-static (computed

6.7.5 C38

Section C: Static analysis to eliminate security concerns.
"To prevent this, we check..".
The hyplet function address is known in advance in the RTOS implementation case.

6.7.6 C39

Section E, Hypervisor based RPC.
We offer a technique to exit endless loops in 6.7.1.

6.8 Paper PVIII: Hypervisor Memory acquisition for ARM

6.8.1 C40

Page 3. The idea of Hypervisor based memory acquisition isn't new.
This paper extends paper PVII, "Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot", and Stüttgen and Cohen (2013) to Volatility and ARMv8.