



**KTH Computer Science
and Communication**

Secure System Virtualization: End-to-End Verification of Memory Isolation

HAMED NEMATI

Doctoral Thesis
Stockholm, Sweden 2017

TRITA-CSC-A-2017:18
ISSN 1653-5723
ISRN-KTH/CSC/A--17/18-SE
ISBN 978-91-7729-478-8

KTH Royal Institute of Technology
School of Computer Science and Communication
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläggges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredagen den 20 oktober 2017 klockan 14.00 i Kollegiesalen, Kungl Tekniska högskolan, Brinellvägen 8, Stockholm.

© Hamed Nemati, October 2017

Tryck: Universitetsservice US AB

Abstract

Over the last years, security kernels have played a promising role in reshaping the landscape of platform security on today’s ubiquitous embedded devices. Security kernels, such as separation kernels, enable constructing high-assurance mixed-criticality execution platforms. They reduce the software portion of the system’s trusted computing base to a thin layer, which enforces isolation between low- and high-criticality components. The reduced trusted computing base minimizes the system attack surface and facilitates the use of formal methods to ensure functional correctness and security of the kernel.

In this thesis, we explore various aspects of building a provably secure separation kernel using virtualization technology. In particular, we examine techniques related to the appropriate management of the memory subsystem. Once these techniques were implemented and functionally verified, they provide reliable a foundation for application scenarios that require strong guarantees of isolation and facilitate formal reasoning about the system’s overall security.

We show how the memory management subsystem can be virtualized to enforce isolation of system components. Virtualization is done by using direct paging that enables a guest software under some circumstances to manage its own memory configuration. We demonstrate the soundness of our approach by verifying that the high-level model of the system fulfills the desired security properties. Through refinement, we then propagate these properties (semi-) automatically to the machine-code of the virtualization mechanism.

An application of isolating platforms is to provide external protection to an untrusted guest operating system to restrict its attack surface. We show how a runtime monitor can be securely deployed alongside a Linux guest on a hypervisor to prevent code injection attacks targeting Linux. The monitor takes advantage of the provided separation to protect itself and to retain a complete view of the guest software.

Separating components using a low-level software, while important, is not by itself enough to guarantee security. Indeed, current processors architecture involves features, such as caches, that can be utilized to violate the isolation of components. In this thesis, we present a new low noise attack vector constructed by measuring cache effects. The vector is capable of breaching isolation between components of different criticality levels, and it invalidates the verification of software that has been verified on a memory coherent (cacheless) model. To restore isolation, we provide a number of countermeasures. Further, we propose a new methodology to repair the verification of the software by including data-caches in the statement of the top-level security properties of the system.

Sammanfattning

Inbyggda system finns överallt idag. Under senare år har utvecklingen av platformssäkerhet för inbyggda system spelat en allt större roll. Säkerhetskärnor, likt isoleringskärnor, möjliggör konstruktion av tillförlitliga exekveringsplatser ämnade för både kritiska och icke-kritiska tillämpningar. Säkerhetskärnor reducerar systemets kritiska kodmängd i ett litet tunt mjukvarulager. Detta mjukvarulager upprättar en tillförlitlig isolering mellan olika mjukvarukomponenter, där vissa mjukvarukomponenter har kritiska roller och andra inte. Det tunna mjukvarulaget minimerar systemets attackyta och underlättar användningen av formella metoder för att försäkra mjukvarulagrets funktionella korrekthet och säkerhet.

I denna uppsats utforskas olika aspekter för att bygga en isoleringskärna med virtualiseringsteknik sådant att det går att bevisa att isoleringskärnan är säker. I huvudsak undersöks tekniker för säker hantering av minnessystemet. Implementering och funktionell verifiering av dessa minneshanteringstekniker ger en tillförlitlig grund för användningsområden med höga krav på isolering och underlättar formell argumentation om att systemet är säkert.

Det visas hur minneshanteringssystemet kan virtualiseras i syfte om att isolera systemkomponenter. Virtualiseringstekniken bakom minnessystemet är direkt paging och möjliggör gästmjukvara, under vissa begränsningar, att konfigurera sitt eget minne. Denna metods sundhet demonstreras genom verifiering av att högnivåmodellen av systemet satisfierar de önskade säkerhetsegenskaperna. Genom förfining överförs dessa egenskaper (semi-) automatiskt till maskinkoden som utgör virtualiseringmekanismen.

En isolerad gästabplikation ger externt skydd för ett potentiellt angriperat gästoperativssystem för att begränsa den senares attackyta. Det visas hur en körningstidsövervakare kan köras säkert i systemet ovanpå en hypervisor bredvid en Linuxgäst för att förhindra kodinjektionsattacker mot Linux. Övervakaren utnyttjar systemets isoleringsegenskaper för att skydda sig själv och för att ha en korrekt uppfattning av gästmjukvarans status.

Isolering av mjukvarukomponenter genom lågnivåmjukvara är inte i sig tillräckligt för att garantera säkerhet. Dagens processorarkitekturen har funktioner, som exempelvis cacheminnen, som kan utnyttjas för att kringgå isoleringen mellan mjukvarukomponenter. I denna uppsats presenteras en ny attack som inte är störningskänslig och som utförs genom att analysera cacheoperationer. Denna attack kan kringgå isoleringen mellan olika mjukvarukomponenter, där vissa komponenter har kritiska roller och andra inte. Attacken ogiltigförklarar också verifiering av mjukvara om verifieringen antagit en minneskoherent (cachefri) modell. För att motverka denna attack presenteras ett antal motmedel. Utöver detta föreslås också en ny metodik för att återvalidera mjukvaruverifieringen genom att inkludera data-cacheminnen i formuleringen av systemets säkerhetsegenskaper.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my main supervisor, Prof. Mads Dam for his encouragement and support, which brought to the completion of this thesis. Mads' high standards have significantly enriched my research.

I am deeply grateful to my colleagues Roberto Guanciale, Oliver Schwarz (you are one of “the best” ;)), Christoph Baumann, Narges Khakpour, Arash Vahidi, and Viktor Do for their supports, valuable comments, and advices during the entire progress of this thesis.

I would also like to thank all people I met at TCS, especially Adam Schill Collberg, Andreas Lindner, Benjamin Greschbach, Cenny Wenner, Cyrille Artho (thanks for reading the thesis draft and providing good suggestions), Daniel Bosk (trevligt att träffas Daniel), Emma Enström, Hojat Khosrowjerdi, Ilario Bonacina, Jan Elffers, Jesús Giráldez Crú, Jonas Haglund (thanks for helping with the abstract), Joseph Swernofsky, Linda Kann, Marc Vinyals, Mateus de Oliveira Oliveira, Mika Cohen, Mladen Mikša, Muddassar Azam Sindhu, Musard Balliu, Pedro de Carvalho Gomes, Roelof Pieters (I like you too :)), Sangxia Huang, Susanna Figueiredo de Rezende, Thatchaphol Saranurak, Thomas Tuerk, and Xin Zhao for lunch company and making TCS a great place.

I am truly thankful to my dear friends Vahid Mosavat, Mohsen Khosravi, Guillermo Rodríguez Cano, and Ali Jafari, who made my stay in Sweden more pleasant.

Last but not the least, huge thanks to my parents, my sister and Razieh for their endless love, moral support and encouragement, without which I would have never been able to complete this important step of my life.

Contents

Contents	vi
I Introduction and Summary	1
Abbreviations	3
1 Introduction	5
1.1 Overview	5
1.2 Thesis Statement	7
1.3 Thesis Outline	9
2 Background	11
2.1 Platform Security	12
2.2 Formal Verification	24
2.3 Summary	36
3 Related Work	37
3.1 Verification of Security-Kernels	37
3.2 Trustworthy Runtime Monitoring	40
3.3 Attack On Isolation	41
3.4 Summary	41
4 Contributions	43
4.1 Summary of Included Papers	43
4.2 Further Publications	48
5 Conclusions	49
5.1 Contribution	49
5.2 Concluding Remarks and Future Work	50

II Included Papers	53
A Provably secure memory isolation for Linux on ARM	55
A.1 Introduction	55
A.2 Related Work	58
A.3 Verification Approach	60
A.4 The ARMv7 CPU	66
A.5 The Memory Virtualization API	70
A.6 Formalizing the Proof Goals	77
A.7 TLS Consistency	83
A.8 Refinement	87
A.9 Binary Verification	88
A.10 Implementation	98
A.11 Evaluation	102
A.12 Applications	103
A.13 Concluding Remarks	107
B Trustworthy Prevention of Code Injection in Linux on Embedded Devices	109
B.1 Introduction	109
B.2 Background	111
B.3 Design	116
B.4 Formal Model of MProsper	118
B.5 Verification Strategy	121
B.6 Evaluation	124
B.7 Related Work	125
B.8 Concluding Remarks	126
C Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures	129
C.1 Introduction	129
C.2 Background	131
C.3 The New Attack Vectors: Cache Storage Channels	133
C.4 Case Studies	139
C.5 Countermeasures	146
C.6 Verification Methodology	154
C.7 Related Work	158
C.8 Concluding Remarks	161
D Formal Analysis of Countermeasures against Cache Storage Side Channels	163
D.1 Introduction	163
D.2 Related Work	165
D.3 Threats and Countermeasures	166

D.4	High-Level Security Properties	168
D.5	Formalisation	169
D.6	Integrity	172
D.7	Confidentiality	180
D.8	Case Study	189
D.9	Implementation	190
D.10	Conclusion	191
D.11	Appendix	194

Bibliography**201**

Part I

Introduction and Summary

Abbreviations

This list contains the acronyms used in the first part of the thesis. The page numbers indicate primary occurrences.

AST Abstract Syntax Tree. 27

BAP Binary Analysis Platform. 26

BIL BAP Intermediate Language. 26

CFG Control Flow Graph. 26

CNF Conjunctive Normal Form. 27

COTS Commercial Off The Shelf. 6

DACR Domain Access Control Register. 20

DMA Direct Memory Access. 15

GDB GNU Debugger. 47

HDM Hierarchical Development Methodology. 39

IOMMU Input/Output Memory Management Unit. 15

IoT Internet of Things. 11

ISA Instruction Set Architecture. 22

LSM Linux Security Module. 21

LTS Labeled Transition System. 31

MILS Multiple Independent Levels of Security. 6

MMU Memory Management Unit. 7

MPU Memory Protection Unit. 14

OS Operating System. 5

RISC Reduced Instruction Set Computing. 23

SGX Software Guard Extensions. 15

SMT Satisfiability Modulo Theories. 27

TCB Trusted Computing Base. 6

TLS Top Level Specification. 34

TOCTTOU Time-Of-Check-To-Time-Of-Use, asdasdasd. 22

VM Virtual Machine. 17

VMI Virtual Machine Introspection. 21

Chapter 1

Introduction

Nowadays, for better or worse, the demand for new (embedded) devices and applications to manage all aspects of daily life from traffic control to public safety continues unabated. However, as reliance on automated sensors and software has improved productivity and people's lives, it also increases risks of information theft, data security breaches, and vulnerability to privacy attacks. This also raises concerns about illicit activities, sometimes supported by governments, to exploit bugs in systems to take over the control of security-critical infrastructures or gain access to confidential information. Series of attacks that hit Ukrainian targets in the commercial and government sectors [200], including power grid and the railway system, are examples reinforcing how vulnerable a society can be to attacks directed at its core infrastructure.

Ever-evolving cyber-attacks are expanding their targets, and techniques used to mount these attacks are becoming increasingly complex, large-scale, and multi-faceted. The advent of new attack techniques in rapid succession and the number of daily uncovered zero-day vulnerabilities [201] are convincing evidences that if the approach to security does not become more formal and systematic, building trustworthy and reliable computing platforms seems as distant as ever.

1.1 Overview

In order to secure a system against attacks, including those that are currently unknown, one needs to consider the security of all layers from hardware to applications available to the end users. Nevertheless, among all constituents, the correctness of the most privileged software layer within the system architecture, henceforth the *kernel*, is an important factor for the security of the system. This privileged layer can be the kernel of an Operating System (OS) or any other software that directly runs on bare hardware, manages resources, and is allowed to execute processor privileged instructions. Therefore, any bug at this layer can significantly undermine the overall security of the system.

It is widely acknowledged that the monolithic design and huge codebase of mainstream execution platforms such as Commercial Off The Shelf (COTS) operating systems make them vulnerable to security flaws [124, 126, 229]. This monolithic design is also what makes integrating efficient and fine-grained security mechanisms into current OSs, if not impossible, extremely hard. The vulnerability of modern desktop platforms to malwares can be attributed to these issues, and it reveals the inability of the underlying kernel to protect system components properly from one another. In order to increase the security and reliability, existing operating systems such as Linux use access control and supervised execution techniques, such as LSM [228] or SELinux [144], together with their built-in process mediation mechanisms. However, while these techniques represent a significant step towards improving security, they are insufficient for application scenarios which demand higher levels of trustworthiness. This is essentially because the kernel of the hosting OS as the Trusted Computing Base (TCB)¹ of the system is itself susceptible to attacks which can compromise these security mechanisms.

A practical solution to increase the system security and to mitigate the impact of any probable malfunctioning is reducing the size and complexity of the kernel. Minimizing the kernel can be done by splitting it into smaller components with restricted² privileges that are isolated from each other and can interact through controlled communication channels. This concept is an old one. Systems based on such a design principle are usually said to implement the Multiple Independent Levels of Security (MILS) philosophy [13]. MILS is a design approach to build high-assurance systems and is the main idea behind the development of *security kernels*.

A security kernel, in the context of this thesis, is an executive that partitions system components into different security classes, each with certain privileges, manages system resources, allocates resources to *partitions*, and controls the interaction of partitions with each other and the outside world. Among others, *microkernels* [142, 107, 133], *separation kernels* [180, 108], and security *hypervisors* [182, 183, 147] are important representatives of security kernels.

Rushby was the first to introduce the notion of separation kernels [180]. A separation kernel resembles a physically distributed system to provide a more robust execution environment for *processes* running on the kernel [180] (cf. Figure 1.1). The primary security property enforced by a separation kernel is *isolation*; i.e., separation of processes in a system to prevent a running process from unauthorized reading or writing into memory space and register locations allocated to other ones. Separation kernels reduce the software portion of the system's TCB to a thin layer which provides the isolation between partitions and implements communication channels.

The reduced TCB minimizes the attack surface of the system and enables the

¹The TCB of a system is the set of all components that are critical to establishing and maintaining the system's security

²The least authority necessary for a component to function correctly.

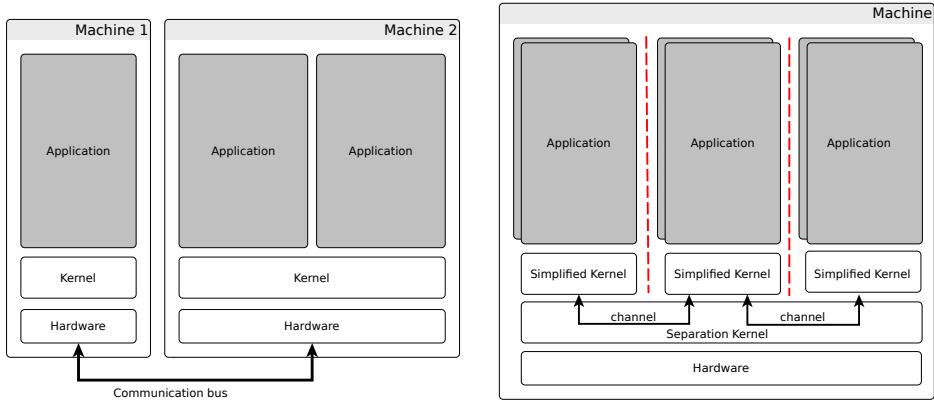


Figure 1.1: Traditional design (left) isolating components using discrete hardware platforms. A separation kernel (right) provides isolation on single processor.

use of rigorous inspection techniques, in the form of a formal proof, to ensure bug-freeness of the partitioning layer. Formal verification lends a greater degree of trustworthiness to the system and ensures that — within an explicitly given set of assumptions — the system design behaves as expected.

Separating components using a low-level system software, assisted with some basic hardware features, reduces overhead in terms of the hardware complexity and increases modularity. However, it is not by itself enough to guarantee security. In fact, current processor architectures involve a wealth of features that, while invisible to the software executing on the platform, can affect the system behavior in many aspects. For example, the Memory Management Unit (MMU) relies on a caching mechanism to speed up accesses to *page-tables* stored in the memory. A cache is a shared resource between all partitions, and it thus potentially affects and is affected by activities of each partition. Consequently, enabling caches may cause unintended interaction between software running on the same processor, which can lead to cross-partition information leakage and violation of the principle of isolation. Therefore, it is essential to consider the impact of such hardware pieces while designing and verifying isolation solutions.

1.2 Thesis Statement

In this thesis, we investigate various aspects of building a provably secure separation kernel using *virtualization technology* and *formal methods*. In particular, we examine techniques related to the appropriate management of the memory subsystem. Once these techniques were implemented and verified, they provide reliable mechanisms for application scenarios that require strong guarantees of isolation and

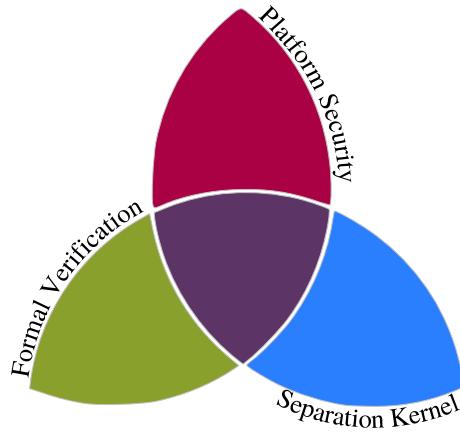


Figure 1.2: Contribution areas.

facilitate formal reasoning about the system functional correctness and its security. We are concerned with understanding how formal verification influences the design of a separation kernel and its trustworthiness, and what the impact of resource sharing on the verification of the system is (cf. Figure 1.2).

We limit our discussion to single-core processors to provide a detailed analysis, rather than dealing with complexities which arise due to the use of multiple cores. More specifically, in this work we tried to address the following issues:

- How can we build a light-weight and performant virtualized memory management subsystem that allows dynamic management of the memory and is small enough to make formal verification possible? This is important since supporting dynamic memory management is essential to run commodity operating systems on separation kernels.
- How can we formally ensure that machine code executing on hardware implements the desired functionalities and complies with the security properties verified at the source-code level? This obviates the necessity of trusting the compilers and provides a higher degree of security assurance.
- How can we take advantage of isolation of system components on a separation kernel to guarantee that only certified programs can execute on the system?
- What are the effects of enabling low-level hardware features such as caches on the security of the system? How can one neutralize these effects to ascertain proper isolation of the system components?

1.3 Thesis Outline

This thesis is divided into two parts. The first part provides the required background to understand the results, and gives a summary of the papers. This part is structured as follows: Chapter 2 discusses the general context and different approaches to formal verification and process isolation, describes the tools we have used and provides the formalization of our main security properties; Chapter 3 gives an overview of related work, including some historical background and recent developments; Chapter 4 describes briefly each of the included papers together with the author's individual contribution; Chapter 5 presents concluding remarks. The second part includes four of the author's papers.

Chapter 2

Background

Embedded computing systems face two conflicting phenomena. First, security is becoming an increasingly important aspect of new devices such as smartphones and Internet of Things (IoT) systems. Secondly, the design of current operating systems and hardware platforms puts significant limitations on the security guarantees that these systems can provide. The problem arises mainly because:

- Commodity operating systems are mostly designed with focus on usability rather than security and inadequately protect applications from one another, in the case of mission-critical scenarios. Therefore, bugs in one application can potentially lead to the compromise of the whole system. This limits the capability of the system to securely execute in parallel applications with different criticality levels.
- Mainstream operating systems are very complex programs with large software stacks. This complexity undermines the system trustworthiness [124, 126, 229] and imposes significant hurdles on building applications with high-criticality on these platforms.
- Current platforms do not provide the end users with any reliable method to verify the identity of applications. For example, a user has no way of verifying if they are interacting with a trusted banking application or with a malicious or compromised program which impersonates the original one.
- Finally, modern processor architectures comprises a wealth of features that, while important for performance, could potentially open up for unintended and invisible paths for sensitive information to escape.

A possible mitigation of these problems is employing special-purpose closed platforms [85], customized for particular application scenarios, e.g., missile controlling systems and game consoles. The design of such closed systems, both at the hardware level and for the software stack, can be carefully tailored to meet required

security guarantees, thus significantly reducing vulnerability to attacks and the cost of formal verification.

Despite multiple security benefits of using closed platforms, in most cases, flexibility and rich functionalities offered by general-purpose open systems make the choice of these systems preferable. In this thesis, we look for a compromise between these two rather conflicting approaches. We try to keep a well-defined structure (i.e., minimal and suitable for a formal analysis) and security guarantees of the closed systems while taking the advantages of using a general-purpose open platform. Our solution is based on a separation kernel that brings together the capabilities of these two systems. Further, we apply mathematical reasoning to show correctness and security of our solution. Even though the focus of the work in this thesis is on design and verification of a system software for embedded devices, namely ARM processors, our results can be adapted to other architectures.

In the following, we give some preliminaries that will be used in this thesis. In particular, Section 2.1 briefly surveys techniques used in practice to provide a trusted execution environment for high-criticality applications on commodity platforms, introduces the concept of security kernel and compare it to other approaches to clarify the advantage of using such kernels. This section also shows how a security kernel can be used to help a commodity OS to protect itself against external attacks and explains why enabling low-level hardware features can potentially affect the security of a system. Section 2.2 introduces the notion of formal analysis and explains techniques that can be used to verify a software. Additionally, in this section, we formalize properties which we have used to analyze the security of a separation kernel and explain our verification methodology.

2.1 Platform Security

2.1.1 Process Protection

Anderson [16] argued that resource sharing is the key cause of many security issues in operating systems, and programs together with their assigned resources must be securely compartmentalized to minimize undesired interaction between applications. The classical method of preventing applications concurrently executing on the same processor from interfering with one another is *process¹ isolation* [16]. Isolation is a fundamental concept in platform security, and it aims to segregate different applications' processes to prevent the private data of a process from being written or read by other ones; the exception is when an explicit communication channel exists. Process isolation preserves the *integrity* of processes and guarantees their *confidentiality*.

Integrity refers to the ability of the system to prevent corruption of information, as both program and data. Corruption can happen either intentionally with the aim

¹A process is an instance of an executable program in the system.

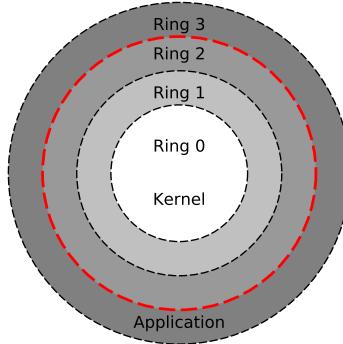


Figure 2.1: Hierarchical protection domains. Ring 0 denotes the most privileged domain.

of benefiting from information rewriting or erasure, or unintentionally, for example, due to hardware malfunction. The integrity property restricts who can create and modify trusted information.

Confidentiality is the property of preventing information disclosure; i.e., making information unreachable by unauthorized actors. This property covers both the existence of information and its flow in the system.

Isolation of processes can be done by hardware or software based solutions; however both these approaches try to limit access to system resources and to keep programs isolated to their assigned resources. In what follows we briefly describe some of the widely applied techniques to achieve isolation.

Memory Management Unit

The notion of isolation in current operating systems is provided at a minimum by abstraction of virtual memory, constructed through a combination of core hardware functionalities and kernel level mechanisms. For all but the most rudimentary processors, the primary device used to enable isolation is the memory management unit, which provides in one go both virtual addressing and memory protection. Operating systems use the MMU to isolate processes by assigning to each a separated virtual space. This prevents errors in one user mode program from being propagated within the system. It is the role of the kernel to configure the MMU to confine memory accesses of less privileged applications. Configurations of the MMU, which are also called page-tables, determine the binding of physical memory locations to virtual addresses and hold restrictions to access resources. Hence page-tables are critical for security and must be protected. Otherwise, malicious processes can bypass the MMU and gain illicit accesses.

To protect security-critical components such as configurations of the MMU, most processors allow execution of processes with different criticality in separate *processor modes* or *protection rings* (c.f. Figure 2.1). On these processors, a privileged program such as the kernel of an OS runs in a special kernel domain, also called kernel space/mode, that is protected from the user specific domains, also referred to as user space/mode, in which less privileged and potentially malicious applications execute.

Some processors such as ARM family CPUs — instead of providing a memory management unit — protect system resources through a Memory Protection Unit (MPU). The MPU is a light weight specialization of the MMU which offers fewer features and is mainly used when the software executing on the platform is far simpler than off-the-shelf operating systems. In contrast to the MMU which uses hardware protection together with the virtual memory capability, the MPU provides only hardware protection over software-designated memory regions.

Process isolation solely based on these approaches has been proven inadequate to guarantee isolation in most cases. Due to the security concerns related to weak isolation, research on platform security has been focused on complementary techniques to strengthen isolation of processes, e.g., *access control* mechanisms, *sandboxing*, and *hardware based* solutions.

Access Control

To improve the basic protection provided by the virtual memory abstraction, traditionally operating systems also use access enforcement methods to validate processes' request (e.g., read, write) to access resources (e.g., files, sockets). Each access control mechanism consists of two main parts: an *access policy store* and an *authorization module*. The access policy describes the set of allowed operations that processes can perform on resources and is specific to each system. An example of such a policy is Lampson's access matrix [137]. At the heart of this protection system is the authorization module which is commonly referred to as *reference monitor*. For a given input request, the reference monitor returns a binary response showing if the request is authorized by the monitor's policy. AppArmor [31] is an example of such an access control mechanism, which is used in some Unix-based operating systems.

Sandbox Based Isolation

Sandboxing, as defined in [218], is the method of encapsulating an unknown and potentially malicious code in a region of memory to restrict its impact on the system state. Accesses of a sandboxed program are limited to only memory locations that are inside the assigned address space, and the program cannot execute binaries not placed in its code segment. In this approach effects of a program are hidden from the outside world. However, for practical reasons sometimes this isolation has to be violated to allow data transmission.

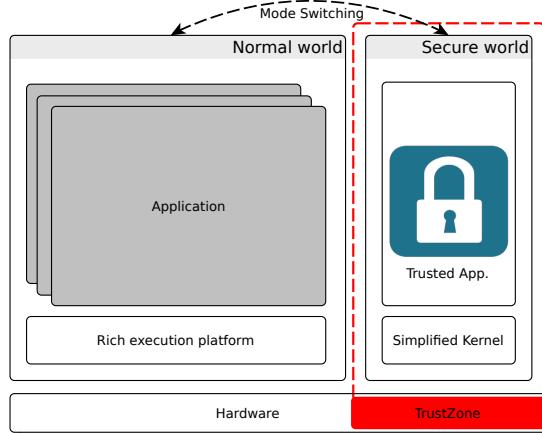


Figure 2.2: ARM TrustZone

Instruction Set Architecture based sandboxing is the technique of controlling activities of a program at the instruction level, through adding instructions to the binary of the program to check its accesses. Software Fault Isolation [218] and Inline Reference Monitors [80] are two examples of sandboxes that are implemented using this technique. Application sandboxing can also be achieved by taking control of the interface between the application and its libraries or the underlying operating system [217]. Further, one can restrict permissions of a program to access system resources using access control mechanisms [52, 50].

Hardware-Extension Based Isolation

Process isolation through features embedded in hardware provides a strong form of separation. These features are either part of the processor or hardware extensions that augment basic protection supplied with the CPU. Examples of these features include IOMMU [35], ARM TrustZone [1], and Intel Software Guard Extensions (SGX) [148, 15]. The Input/Output Memory Management Unit (IOMMU) is a hardware extension to control memory accesses of I/O devices. The IOMMU prevents malicious devices from performing arbitrary Direct Memory Access (DMA) operations and can be used to isolate device drivers.

TrustZone (c.f. Figure 2.2) is a set of security extensions added to some ARM architecture CPUs to improve the system security. TrustZone creates a secure execution environment, also called *secure world*, for software that must be isolated from less critical components. These extensions allow the processor to switch between two security domains that are orthogonal to the standard capabilities of the CPU. TrustZone can be used to execute an unmodified commodity OS in the less secure domain, and to run security-critical subsystems (e.g., cryptographic algorithms)

inside the secure domain. This hardware feature guarantees that the critical subsystems will remain safe regardless of malicious activities influencing the system outside the *secure world*.

Intel's SGX are extensions to x86 processors that aim to guarantee integrity and confidentiality of the code executing inside SGX's secure containers, called *enclaves*. The main application of the SGX is in *secure remote computation*. In this scenario the user uploads his secret data and computation method into an enclave and SGX guarantees the confidentiality and integrity of the secret user data while the computation is being carried out.

2.1.2 Strong Isolation and Minimal TCB

Process isolation is essential to platform security. Nevertheless, in today's increasingly sophisticated malware climate, methods commonly used to achieve isolation, such as the ones we have discussed above, are gradually proving insufficient. The main typical problems related to the kernel-level solutions like access control and sandboxing mechanisms is that they enlarge the trusted computing base of the system and their trustworthiness depends heavily on the characteristics of the underlying OS. Language-based techniques [184] are mostly experimental, and tools available to enforce them are research prototypes and not applicable to large-scale real-world systems. Using type systems [184] to enforce isolation is not feasible since most system software mix C (which is not generally a type-safe programming language [156]) with assembly (which does not support fine-grained types). Hardware-based solutions are helpful, but they do not solve the problem entirely either. They increase the bill-of-materials costs, and (potential) bugs in their implementation can be exploited by attackers to violate isolation enforced using these features [193, 64, 120, 197, 95].

Isolation can be most reliably achieved by deploying high- and low-criticality components onto different CPUs. This, however, leads to higher design complexity and costs. These make such an approach less appealing and emphasize the significance of software based solutions again. Security kernel (e.g., microkernels [142, 107, 133], separation kernels [174], and hypervisors [26, 189]) are recognized as practical solutions to mitigate the problems of the aforementioned techniques that bring together the isolation of dedicated hardware and the enjoyments of having a small TCB.

Microkernels

The primary objective of microkernels is to minimize the trusted computing base of the system while consolidating both high- and low-criticality components on a single processor. This is usually done by retaining inside the most privileged layer of the system only those kernel services that are security-critical such as memory and thread management subsystems and inter-process communication. Other kernel level functionalities can then be deployed as user space processes with limited access

rights. Using a microkernel the user level services are permitted to perform only accesses that are deemed secure based on some predefined policies.

A fully-fledged operating system can be executed on a microkernel by delegating the process management of the hosted OS completely to the microkernel (e.g., L⁴Linux) through mapping the guest’s threads directly to the microkernel threads. However, this generally involves an invasive and error-prone OS adaptation process. Alternatively, the microkernel can be extended to virtualize the memory subsystem of the guest OS (e.g., using shadow-paging [8] or nested-paging [39]).

Separation Kernels

Separation kernels are software that enforce separation among system components and are able to control the flow of information between partitions existing on the kernel [180]. Programs running on separation kernels should behave equivalently as they were executing on distributed hardware. Communication between partitions is only allowed to flow as authorized along well-defined channels. Similar to microkernels, separation kernels implement the MILS philosophy [13].

The idea of separation kernels is primarily developed to enforce (security-) isolation and many such kernels do not support essential functionalities to host a complete operating system including device drivers and file systems. The commonly used technique to compensate this shortcoming is virtualization. The advent of virtualization technologies provides significant improvements in efficiency and capabilities of the security kernel. Virtualization allows building high-assurance systems having the same functionalities as the commodity execution platforms. In this thesis, we use virtualization as an enabler for isolation and to implement a separation kernel capable of hosting a complete operating system.

Virtualization

Virtualization, as it is used in this thesis, is the act of abstracting the underlying hardware to multiplex resources among multiple guests and security-check accesses to system resources before executing them. The virtualization layer, which is also called *hypervisor*, executes at the most privileged mode of the processor and can interpose between the guests and hardware. This enables the hypervisor to intercept guests’ sensitive instructions before being executed on hardware. The complete mediation of events allows the creation of isolated partitions (sometimes referred to as Virtual Machine (VM)) in which applications with an unknown degree of trustworthiness can execute safely. Platform virtualization can be done in many ways, two predominant approaches to virtualization are: *full virtualization* and *paravirtualization*.

- Full virtualization is the technique of providing a guest software with the illusion of having sole access to the underlying hardware; i.e., the virtualization layer is transparent to the guest. In this approach, the hypervisor

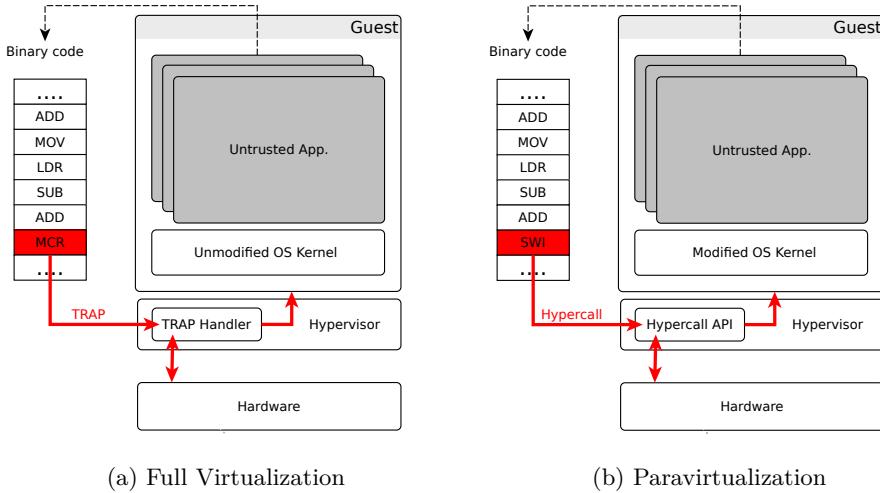


Figure 2.3: Different virtualization approaches.

resides at the highest privilege level and controls the execution of sensitive (or privileged) instructions. Whenever the deprivileged guest software tries to execute one of these sensitive instructions, the execution “traps” into the virtualization layer and the hypervisor emulates the execution of the instruction for the guest (c.f. Figure 2.3a). The advantage of using this approach is that binaries can execute on the hypervisor without any changes, neither the operating systems nor their applications need any adaptation to the virtualization layer. However, this technique increases complexity of the hypervisor design and introduces relatively high performance penalties.

- In contrast, in a paravirtualized system, the guest is aware of the virtualization layer. Using this technique, the execution of sensitive instructions in the guest software is replaced with an explicit call, by invoking a *hypcall* or a *software interrupts*, to the hypervisor (c.f. Figure 2.3b). Each hypcall is connected to a handler in the hypervisor which is used to serve the requested service. Paravirtualization is proven to be more performant. However, it requires adaptation of the guest to the interface of the hypervisor, which can be a very difficult task.

Paravirtualization can be implemented using the same hardware features that operating systems use. Nevertheless, efficient full virtualization of the system usually requires support from hardware primitives such as the processor or I/O devices. Essential to full virtualization is the reconfiguration of the guests' privileges so that any attempt to execute sensitive instructions traps into the hypervisor. This en-

tails emulation of hardware functionalities, such as interrupt controller, within the hypervisor to allow execution of hosted software inside partitions.

Modern processors provide features that can be used to simplify hypervisors design and increase their performance, e.g., extended privilege domains and 2-stage MMUs. However, since our main goal is formal verification of the virtualization software we intentionally avoid using these features, which otherwise complicate formal analysis by shifting the verification burden from a flexible software to the fixed hardware.

In order for a hypervisor to host a general-purpose OS, it is generally necessary to allow the guest software to dynamically manage its internal memory hierarchy and to impose its own access restrictions. To achieve this, a mandatory security property that must be enforced is the complete mediation of the MMU settings through virtualizing the memory management subsystem. In fact, since the MMU is the key functionality used by the hypervisor to isolate the security domains, violation of this security property enables an attacker to bypass the hypervisor policies which could compromise the security of the entire system. This criticality is also what makes a formal analysis of correctness a worthwhile enterprise.

Widely adopted solutions to virtualize the memory subsystem are shadow paging, nested paging, microkernels, and *direct paging*. A hypervisor implemented using shadow paging keeps a copy of the guest page-tables in its memory to perform (intermediate-) virtual to physical address translation. This copy is updated by the hypervisor whenever the guest operates on its page-tables. Nested paging, is a hardware-assisted virtualization technology which frees hypervisors from implementing the virtualization mechanism of the memory subsystem (e.g., [117, 146]). Direct paging was first introduced by Xen [26] and is proved to show better performance compared to other techniques. In paper A we show a minimal and yet verifiable design of the direct paging algorithm and prove its functional correctness and the guarantees of isolation at the machine code level.

PROSPER kernel Our implemented separation kernel is called PROSPER kernel, which is a hypervisor developed using the paravirtualization technique to improve the security of embedded devices. The hypervisor runs bare bone in the processor’s most privileged mode, manages resource allocation and enforces access restrictions. Implementation of the hypervisor targets BeagleBoard-xM and Beaglebone [60] equipped with an ARM Cortex-A8 processor (with no hardware virtualization extensions support) and allows execution of Linux (kernel 2.6.34 and 3.10) as its untrusted guest. Both user space applications and kernel services (or other trusted functionalities) on the PROSPER kernel execute in unprivileged mode. To this end, the hypervisor splits user mode in two virtual CPU modes, namely *virtual user mode* and *virtual kernel mode*, each with their own execution context. The hypervisor is in charge of controlling context switching between these modes and making sure that the memory configuration is setup correctly to enable separation of high- and low-criticality components. In ARM these virtual modes can be im-

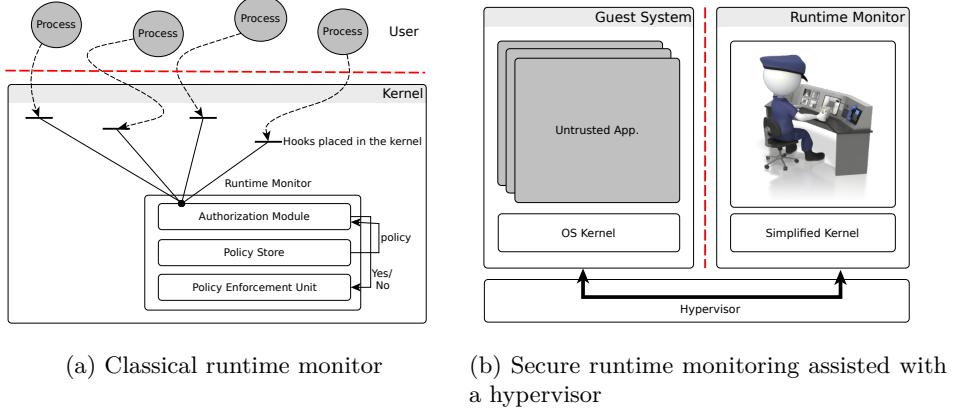


Figure 2.4: Runtime monitor.

plemented through “domains”. These domains implement an access control regime orthogonal to the CPU’s execution modes. We added a new domain for the hypervisor to reside in, and with the help of MMU configurations and the Domain Access Control Register (DACR), the hypervisor can set the access control depending on the active virtual guest mode.

The PROSPER kernel² has a very small codebase, and its design is kept intentionally simple (e.g., the hypervisor does not support preemption) to make the formal verification of the hypervisor affordable.

2.1.3 Secure Runtime Monitoring

The increasing complexity of modern computing devices has also contributed to the development of new vulnerabilities in the design and implementation of systems. Among dangerous vulnerabilities are those that enable an adversary to impersonate trusted applications by either injecting malicious binaries into the executable memory of the applications or causing anomalies in the system control flow to execute arbitrary code on target platforms.

The success of these attacks mostly hinges on unchecked assumptions that the system makes about executables in the memory. Therefore, countermeasures against vulnerabilities of these types include certifying all binaries that are safe to execute, and employing a monitoring mechanism which checks events at runtime to prevent the execution of uncertified programs. A runtime monitor is a classical access enforcement method. A monitor checks validity of the requests to execute binaries through placing hooks (e.g., by patching *system calls*) that invoke the monitor’s authorization module (c.f. Figure 2.4a). The main application domain for a

²A full virtualization variant of the PROSPER kernel was also developed by HASPOC project [32], which supports additional features such as multicore support and secure boot.

runtime monitor is inside a rich execution environment such as a COTS OS, where the monitor runs in parallel with other applications in the system and enforces some security policy.

Essential to making the monitoring process trustworthy and effective is to provide the monitor with a complete view of the system and to make it tamper resistant. Linux Security Module (LSM) [228] is an example of such a monitoring mechanism that is integrated into the Linux kernel. However, kernel-level techniques are not trustworthy as the hosting OS is itself susceptible to attacks [124].

An interesting use-case scenario for isolation provided by a security kernel is when the trusted isolated components are used as an aid for a commodity operating system to restrict the attack surface of the OS. In a virtualized environment the monitoring module can be deployed in a different partition (c.f. Figure 2.4b). Since the hypervisor runs in most privileged mode, it has full control over the target operating system and can provide the monitor with a complete view of the events happening in the system. Such a virtualization assisted monitoring was first introduced by Garfinkel and Rosenblum [86]. Virtual Machine Introspection (VMI), in the terminology of [86], places the monitoring subsystem outside of the guest software, thus making the monitoring module tamper proof. A further advantage of VMI-based solutions is that access enforcement can be done based on information retrieved directly from the underlying hardware, which can not be tampered by an attacker. Security mechanisms that rely on the ability of observing the system state can also benefit from VMI's properties, e.g., isolation.

2.1.4 Side-Channel Attacks

Despite ongoing efforts, developing trusted unbypassable mechanisms to achieve isolation remains a challenge. The problem arises due to the design of current hardware platforms and operating systems. Contemporary hardware platforms provide a limited number of resources such as caches that are shared among several processes by operating systems to enable multitasking on the same processor. While resource sharing is fundamental for the cost-effective implementation of system software, it is essential to do so carefully to avoid initiating unintentional channels which may lead to the disclosure of sensitive information to unauthorized parties. This raises the potential of attack vectors that are not specified by the system specification and some of which are available for user applications to exploit.

Broadly speaking, leakage channels are classified, based on the threat model, into two types:

- (i) *Covert-channels* that are channels used to deliberately transfer secret information to parties not allowed to access it by exploiting hidden capabilities of system features, and
- (ii) *Side-channels* that refer to paths, which exist accidentally to the otherwise secure flow of data, for sensitive information to escape through.

In covert-channel attacks, both sides of the communication are considered malicious. However, in side-channels attacks, only the receiver has malicious intents and tries to get access to secret information through measuring (unintended) side effects of victim computations. Therefore, in COTS system software, we are mostly interested in studying side-channels.

Side-channels can be further subdivided into two groups, namely *storage channels* and *timing channels*. Storage channels, in the current usage, are attacks conducted by exploiting aspects of the system that are directly observable by the adversary, such as values stored in memory locations accessible by the attacker or registers content. In contrast to storage channels, timing attacks rely on monitoring variations in execution time to discover hidden hardware state. Timing channels are, in general, believed to have severe impact and they can occur even when the capability of the attacker to observe system resources is fully understood. However, the noise introduced by actuators operating on the system usually makes timing analysis hard.

One very important representative of side-channels are attacks based on measuring effects of caching on system performance. Caches are hardware components that are widely adopted in computing devices and used to provide quicker response to memory requests to avoid wasting precious processor cycles. Caches can hold recently computed data, not existing in the main memory, or they can be duplicates of original values in the memory. For each memory access, if the requested data is in the cache (cache hit) the request can be served by simply reading the cache line containing data. If data is not present in the cache (cache miss), it has to be recomputed or loaded from its original storage location. The MMU controls accesses to caches, that is, it checks if data can be read from or written into the cache.

While enabling caches is important for performance, without proper management they can be used to extract sensitive information. Cache timing side-channels are attack vectors that have been extensively studied, in terms of both exploits and countermeasures, cf. [198, 162, 224, 129, 90, 55]. However, cache usage has pitfalls other than timing differentials. For instance, for the ARMv7 architecture, memory coherence may fail if the same physical address is accessed using different cacheability attributes. This opens up for Time-Of-Check-To-Time-Of-Use, asdas-dasd (TOCTTOU)³- like vulnerabilities since a trusted agent may check and later evict a cached data item, which is subsequently substituted by an unchecked item placed in the main memory using an uncacheable alias. Moreover, an untrusted agent can similarly use uncacheable address aliasing to measure which lines of the cache are evicted. This results in storage channels that are not visible in information flow analyses performed at the Instruction Set Architecture (ISA) level.

In practice, chip and IP manufacturers provide programming guidelines that

³Time-Of-Check-To-Time-Of-Use is a class of attacks mounted by changing the victim system state between the checking of a (security related) condition and the use of the results of that check.

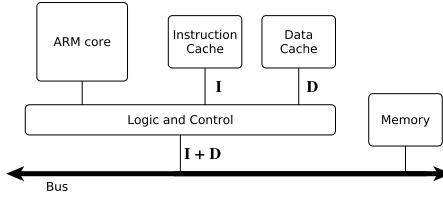


Figure 2.5: Harvard architecture with separate data and instruction caches.

guarantee memory coherence, and they routinely discourage the use of mismatched memory attributes such as cacheability. However, in rich environments like hypervisors or OSs, it is often essential to delegate the assignment of memory attributes to user processes that may be malicious, making it difficult to block access to the vulnerable features.

There are more side-channel attacks that deserve further studies, for instance, attacks based on analysis of power, electromagnetic patterns, acoustic emanations. However, exploring their impact on the system security is out the scope of this thesis. Note also, in contrast to timing and storage channel attacks which can be conducted remotely [46] these attacks need physical access to the victim machine. We are mainly interested in improving aspects of platform security that are relevant to the memory management subsystem. In particular, in paper C we use the L1 data-cache to create low noise storage channels to break isolation between system components. Moreover, we show how these channels can be neutralized by exploiting proper countermeasures.

2.1.5 ARM Architecture

The ARM family processors are based on Reduced Instruction Set Computing (RISC) architecture. The RISC architecture aims to provide a simple, yet powerful set of instructions that are able to execute in a single CPU cycle. This is achieved mostly by placing greater demands on the compiler to reduce the complexity of instructions that hardware executes. In this thesis, we consider Harvard implementations of the ARM cores. The Harvard architecture uses separate buses, and consequently separates caches, for data and instructions to improve performance (cf. Figure 2.5).

The ARM architecture can operate in several execution modes; e.g., ARMv7 has seven modes. Among them, unprivileged user mode is used to execute user space applications, and the others are protected modes each having specific interrupt sources and are reserved for the privileged operations. Privileged modes are accessible through the interrupt *vector table*, which is a table associating interrupt handlers with the corresponding interrupt request.

ARM processors are *load-store* architectures; i.e., the processor operates on data

held in registers. Registers are hardware stores that act as the fast local memory and hold both data and addresses. Among others, general-purpose registers are accessible in all modes and of which the register 15 is the *program counter* and contains the address of the next instruction that CPU will execute, the register number 14 is called *link register* and holds the return address of function calls, and the register 13 is used as the *stack pointer*. In addition to the general-purpose registers, ARM processors also include a number of *control registers* that are used to determine the current execution mode, the active page-table, and to control context switching between modes.

2.2 Formal Verification

The increasing importance of security kernels in system security makes them an interesting target for attackers. This emphasizes the significance of applying formal methods to verify the correctness and isolation guarantees of these kernels. A key research contribution of this work is the formal verification of the PROSPER kernel. We discuss our verification approach in papers A,B,D and provide more details on properties that we have verified. In this section, we give a short exposition of our verification methodology and describe tools that are involved in this exercise.

Establishing trust on software can be achieved in a number of ways. We can make sure that the program design is fully understood and experienced developers are chosen to implement it; we can conduct an extensive software testing procedure to check the quality of the program and its conformance with the system requirements; etc. Unfortunately, while these techniques are helpful to implement high-quality programs with fewer bugs, still we keep finding errors in extensively adopted code, such as the binary search algorithm of the Java API [160] and the open-source cryptographic software library OpenSSL [105, 106]. OpenSSL is behind many secure communication protocols over the Internet and the bug, dubbed Heartbleed, discovered in this cryptographic library could seriously compromise the security of systems using this protocol. Such incidents underline the need of using more rigorous methods to verify the trustworthiness of programs that are security/safety-critical.

Over the last decades, formal verification has emerged as a powerful tool to provide enhanced trustworthiness to systems software like hypervisors, microkernels, and separation kernels [133, 10, 225, 71, 108, 233, 199, 97]. Formal verification provides strong guarantees backed by mathematical proofs that all behaviors of a system meet some logical specification.

There exist several approaches to formal verification with various degrees of automation. For example, *model checking* [79] is a method of automatically proving the correctness of a program based on a logical specification, which expresses certain (temporal) properties of the program such as termination. While model checking is a widely adopted verification approach in industry, it suffers from the *state space*

explosion problem⁴, and its use is limited to prove properties about programs with a finite state space. In what follows we briefly describe the techniques used to formally verify computer programs most relevant to the work presented in this thesis.

2.2.1 Static Program Analysis

Program analysis is the process of examining a program to find bugs and to detect possible misbehavior. In general, research in program verification categorizes this process along two dimensions: dynamic vs. static, and binary vs. source [185]. In static analysis, reasoning is done without actually running programs to determine their runtime properties through analyzing the code structure. This is in contrast to the dynamic approach which verifies a program by inspecting values assigned to variables while the program executes.

The main idea underlying static verification is to specify properties of programs by some *assertions* and to prove that these assertions hold when the execution reaches them. Each assertion is a *first-order* formula constructed using the program's constants, variables, and function symbols, and it describes logical properties of program variables. In this approach, a program is a sequence of *commands* C (or *statements*). The correctness condition of each command is described by an annotation of the form $\{P\} C \{Q\}$, also called a *Hoare-triple* [111]. We say the command C is *partially* correct if whenever C is executed in a state satisfying the *precondition* P and if C terminates, then the state in which the execution of C terminates must meet the *postcondition* Q . By extending this notion, using an inference system, the entire program can be verified using this technique. The inference system is specific to the language of the program's source-code and consists of a set of axioms and rules that allow to derive and combine such triples based on the operational semantics of the language. Note that to prove the *total correctness* of the program, proving that execution will eventually terminate is an additional proof obligation.

For a given postcondition, rules of this inference system also allow computing a *weakest (liberal) precondition*. The weakest precondition (wp) computed using this method can be utilized to verify the triple $\{P\} C \{Q\}$ by checking the validity of a first-order predicate $P \Rightarrow wp(C, Q)$ [41]. Such a predicate often (e.g., if it is quantifier-free) can be resolved using an SMT solver.

Binary Verification

Verifying programs at the level of their source-code, while necessary, is not sufficient on its own. Indeed, most program verification (e.g., verification of a hypervisor) should reach the binaries (or machine-code) of programs, as *object code* that ultimately will execute on hardware. Machine-code needs to be examined to ensure

⁴State space explosion refers to the problem that the memory needed to store the states required to model a system exceeds the available memory.

that properties established at the source-code level hold for programs binary as well.

Binary analysis has its roots in work published by Goldstein and von Neumann [93], where they studied specification and correctness of machine-code programs. Later, Clutterbuck and Carré [54] stressed the significance of formal analysis at the binary level and applied Floyd-Hoare-style verification condition generator to machine codes. Similarly, Bevier [38] showed in his PhD thesis how the kernel of an operating system can be verified down to its binary. Myreen [155] automates the whole binary verification process inside the HOL4 theorem prover [114] by introducing a proof-producing decompilation procedure to transform machine codes to a function in the language of HOL4, which can be used for reasoning.

At the binary level, machine state contains few components (e.g., memory and some registers and status bits), and instructions perform only very minimal and well-defined updates [154]. Analysis of machine codes provides a precise account of the actual behavior of the code that will execute on hardware. Programs' behavior at the level of source-code is generally undefined or vaguely defined; e.g., enabling some aggressive optimization in C compilers can lead to missing code intended to detect integer overflows [220]. Moreover, low-level programs mix structured code (e.g., implemented in C) with assembly and use instructions (e.g., mode switching and coprocessor interactions) that are not part of the high-level language. This makes it difficult to use verification tools that target user space codes. A further advantage of using the binary verification approach is that it obviates the necessity of trusting compilers.

Despite the importance of machine-code verification, it suffers from some problems. Binary codes lack abstractions such as variables, types, functions, and control-flow structure, which makes it difficult to use static analysis techniques to examine machine codes. Furthermore, the indirect jumps existing at the binary level prevent constructing a precise Control Flow Graph (CFG) of programs, which are often used by static analyses. Tackling these sorts of problems researchers have been focused on developing techniques to over-approximate the CFG of programs [25] and to generate an intermediate representation of binaries (or IL) [45, 47]. The IL representation is useful since it helps to mitigate complexities of modern instruction sets and to restore required abstractions. Paper A elaborates on our solution to produce a certified IL representation of the PROSPER kernel and shows how we resolved indirect jumps at the kernel's binary to facilitate machine-code verification.

Binary verification can be automated to a large extent, for example, Bitblaze [196] and Binary Analysis Platform [47] are tools developed to automate verification of functional and safety properties at the machine-code level.

Binary Analysis Platform

The Binary Analysis Platform (BAP) [47] (cf. Figure 2.6) is a framework for analyzing and verifying binary codes. The BAP front-end includes tools to lift input binaries to the BAP Intermediate Language (BIL). BIL provides an architecture

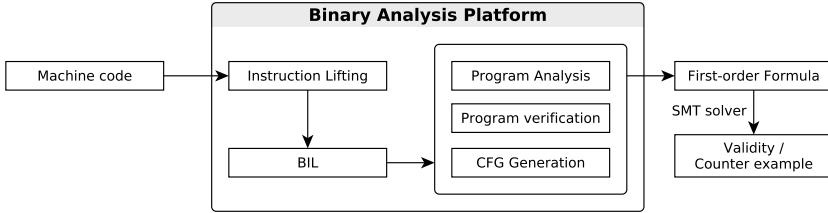


Figure 2.6: Binary Analysis Platform work-flow

independent representation which exposes all the side-effects of machine instructions. BIL code is represented as an Abstract Syntax Tree (AST) that can be traversed and transformed using several methods to perform desired analysis. The back-end of BAP supports utilities to construct and analyze control-flow graphs and program dependence graphs, to perform symbolic execution and to compute and verify contracts by generating the weakest preconditions. The weakest precondition algorithm of BAP provides an effective method to speed up the verification of loop-free assembly fragments, by reducing the problem of verifying Hoare-triples to proving a first-order formula. This formula can in many cases be validated by an external SMT solver (e.g., [74]).

Satisfiability Modulo Theories Solver

Deciding whether a formula that expresses a constraint⁵ has a solution (or a model) is a fundamental problem in the theoretical computer science. There are several problems that can be reduced to constraint satisfiability, including software and hardware verification, type inference, static program analysis, etc. Many of these problems, if expressed using first-order Boolean formulas, can be solved by a *Boolean Satisfiability* (a.k.a. SAT) solver.

A SAT solver determines whether a formula over boolean variables can be made true. For a first-order formula, this amounts to finding an interpretation of its variables, function and predicate symbols which makes the formula true. Generalizing concept of SAT solving, solvers for Satisfiability Modulo Theories (SMT) can also be used to decide satisfiability of boolean formulas. SMT solvers allow to include domain-specific theorems (e.g., real numbers or arrays theories) into the reasoning. This makes SMT solvers more efficient for propositional formulas, but restricts application of these solvers to more specific areas.

2.2.2 Interactive Theorem Proving

Interactive theorem provers, or proof-assistants, are computer programs commonly utilized as an aid for the human user to write and machine-check formal proofs.

⁵Constraints are formulas in Conjunctive Normal Form (CNF).

Interactive provers rely on hints given by the user in constructing proofs rather than generating proofs fully automatically. In this approach, the user specifies the proof-structure and provides some guidance communicated via a domain-specific language to the prover, while the machine checks the proofs and uses the provided hints to automate the process as much as possible. The automation can be achieved by proving proof slices (i.e., sub-goals) that are automatically inferable. In contrast to model checking, theorem proving can be used to verify programs with probably infinitely many states and is the leading approach to deal with new verification challenges, such as verification of system software.

HOL4 Theorem Prover

Among the most popular interactive theorem provers are Isabelle/HOL [159], HOL4 [114], and Coq [61]. HOL4 is a LCF-style [94, 168] proof assistant for Higher-Order-Logic built on a minimal proof kernel that implements the axioms and basic inference rules. Higher-Order-Logic is an extension of first-order logic with types and quantification over functions. HOL4 uses Standard ML as its meta-language⁶ and provides a rich environment with a variety of libraries and theories to prove theorems and to implement proof tools. False statements have no possibility of being proved in HOL4. This is coded through the ML type system to force all proofs to pass the logical kernel of HOL4.

A user of HOL4 has the possibility of steering the system via a number of constructs; namely (i) proof *tactics*, which reduce a *goal* (i.e. theorem to be proved) to simpler subgoals and are used to automate the process of theorem proving, (ii) proof *rules*, which can be used to transform theorems to new ones, (iii) *conversions*, which convert a logical expression into a theorem that establishes the equivalence of the initial expression to the one computed by the applied conversion, and (iv) custom built-in ML programs. HOL4 uses backward reasoning to prove goals. Using this technique, the input theorem is transformed into simpler subgoals by applying proper tactics. Generated subgoals can then be either directly discharged using a simplifier or will be further split into simpler subgoals until they are discharged.

2.2.3 Security Properties

In this subsection, we turn to formalize properties we have used to analyze the security of a hypervisor. The hypervisor plays the role of a security kernel that provides a minimal software base to enforce separation and control the flow of information within a single-core processor. Heitmeyer et al. [108] showed that *data-separation* and *information flow security* properties are strong enough to demonstrate isolation, modulo the explicit communication link, between partitions executing on the hypervisor.

⁶The language used to implement the prover itself.

Data Separation

Informally, separation means prohibiting processes running in one partition from encroaching on protected parts of the system, such as the memory allocated to other guests or the hypervisor internal data-structures. Separation, as discussed in Subsection 2.1.1, can be expressed in terms of two properties, namely integrity and confidentiality or *no-exfiltration* and *no-infiltration* in the terminology of [108].

In the following, we first present formalization of the data-separation property as presented in [108]. Then, we instantiate the separation property with a model analogous to the model of our target platform to get properties similar to the ones we have shown in papers A, B, and D. To this end, we start with a gentle description of notions that are needed to understand the formalism.

We assume the behavior of the kernel is modeled as a state machine defined using a set of states $\sigma \in \Sigma$, an initial state σ_0 , an input alphabet E , and a transition function $T : \Sigma \times E \rightarrow \Sigma$. Each partition is assigned an identifier i and a dedicated region of the memory M_i marked with the partition identifier⁷. The memory of partitions is further subdivided into two parts: (i) a “data memory” area $M_{\{i,d\}}$ which contains all the data-structures belonging to the partition i , and (ii) a number of “input and output buffers” $B_i \in M_{\{i,b\}}$ to communicate with other partitions. The input alphabet consists of a number of internal actions $e_{in} \in D$ and external actions $e_{ext} \in P$. The internal actions are used to invoke a service handler or to manipulate the data memory, and external actions are those that can be executed by fellow partitions or the kernel and have access to the communication buffers. We use D_i (P_i) to denote the internal (external) actions of the partition i and D (P) is the union of the internal (external) actions of all the existing partitions in the system. Moreover, the transition function T transforms the system states by consuming the input events. Having defined this machinery, we can now proceed to give the formal account of no-exfiltration and no-infiltration properties.

No-exfiltration guarantees the integrity of resources not allocated to the active partition (i.e., protected data). This property is defined in terms of the entire memory M , including the data memory and I/O buffers of all partitions, and says that: for every partition i , event $e \in D_i \cup P_i$, and states σ and σ' such that σ' is reachable from σ by the event e , if a transition from σ to σ' changes the content of a memory location m , then m is inside a memory region that is modifiable by i .

Property 1. (*No-exfiltration*) *For all partition i , states σ and σ' in Σ , event $e \in D_i \cup P_i$, and memory location $m \in M$ such that $\sigma' = T(\sigma, e)$, if $m_\sigma \neq m_{\sigma'}$, then $m \in M_i$.*

On the other hand, *no-infiltration* enforces confidentiality by ensuring that data processing of a partition is free of any influences from “secret” values stored in

⁷While in [108] an additional memory area is defined that is shared among partitions, we skip presenting it here to simplify definition of the main properties. This also allows us to combine no-exfiltration with the kernel integrity property of [108].

resources that are inaccessible to the partition. No-infiltration is a 2-safety property [53, 202] and requires reasoning about two parallel executions of the system. This property requires that for every partition i , event $e \in D \cup P$, and states σ_1 , σ_2 , σ'_1 , and σ'_2 such that σ_2 and σ'_2 are, respectively, reachable from σ_1 and σ'_1 , if two executions of the system start in states having the same value in m (which is located inside memory of the partition i), after the event e , the content of m should be changed consistently in both the final states.

Property 2. (*No-infiltration*) *For all partition i , states σ_1 , σ_2 , σ'_1 , and σ'_2 in Σ , and event $e \in D \cup P$, such that $\sigma_2 = T(\sigma_1, e)$ and $\sigma'_2 = T(\sigma'_1, e)$, if for all $m \in M_i$, $m_{\sigma_1} = m_{\sigma'_1}$ then it must hold that for all $m \in M_i$, $m_{\sigma_2} = m_{\sigma'_2}$.*

The no-exfiltration and no-infiltration properties impose constraints on the behavior of the active partition. On models allowing communication, an additional property would be needed to restrict effects of the communication protocol on the partitions' state. *Separation of control* expresses how information can flow through input/output buffers. In particular, this property says that the external events are only allowed to change the buffers of the receiving partition, and this write should not modify the private data memory of the receiver.

Property 3. (*Separation of Control*) *For all partitions i , i' and i'' , states σ and σ' in Σ , and event $e \in D \cup P$ such that i' and i'' are the identifiers of the active partitions in σ and σ' respectively and $\sigma' = T(\sigma, e)$, if $i \neq i'$ and $i \neq i''$ then for all $m \in M_{\{i, d\}}$, $m_{\sigma'_1} = m_{\sigma'_2}$.*

This last property is not covered in this thesis. However, the writer co-authored the paper, “*Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel*” [71], which presents the verification of the (simplified) PROSPER kernel for partitions communicating through *message boxes* (a variant of input/output buffer). A property which subsumes Separation of Control and is proved for the kernel in this verification exercise is to show that, while the hypervisor is able to change message boxes, it cannot modify the other state components belonging to guests.

The hypervisor we use here has the full control of the platform and provides for each guest a virtual space in much the same way that a process runs in an OS-provided virtual memory area. The hypervisor hosts a few guests each assigned statically allocated and non-overlapping memory regions. Execution of the guests is interleaved and controlled by the hypervisor. That is, at the end of a specific time slice (when a timer interrupt happens) or when the active guest explicitly invokes a functionality of a fellow partition (through explicitly invoking a hypercall), the hypervisor suspends the active guest and resumes one of idle guests (e.g., the one which hosts the functionality invoked by the active partition). The guests are virtualization aware (paravirtualized) software running entirely in unprivileged user mode and ported on the hypervisor to use the exposed APIs. Moreover, the only

software that executes in privileged mode is the hypervisor and its execution cannot be interrupted, i.e., the hypervisor does not support preemption.

The models of our system in this thesis are (ARM-flavoured) instruction set architecture models. The memory in these models is partitioned into two segments, namely a code segment and a data segment. Executable code (i.e., a sequence of instructions) resides in the memory code segment, and the processor fetches and executes instructions according to the value of a hardware store, called *program counter*. In such models the data memory $M_{\{-,d\}}$ is the aggregation of both the code and data memories, and events can be replaced by execution of instructions, which makes the notion of internal/external events superfluous. Also, in this section, we assume that the partitions on the hypervisor are non-communicating.

System state A state $\sigma \in \Sigma$ in our model consists of the values of various machine (i.e., the hardware platform) components such as registers (including both general-purpose and control registers), memory, and caches.

Execution An execution in this model is defined as a sequence of configurations from the state space Σ . We represent the transition of states using a deterministic Labeled Transition System (LTS) $\rightarrow_m^n \subseteq \Sigma \times \Sigma$, where $n \in \mathbb{N}$ is the number of taken steps, and $m \in \{0, 1\}$ determines the execution mode (i.e., either privileged mode 1 or unprivileged mode 0). Then, for states σ and σ' a transition from σ to σ' can be defined as the following: if the number of taken steps is greater than zero $n > 0$ then there exist an intermediate state σ'' which is reachable from σ in $n - 1$ steps and σ' is the state immediately after σ'' , otherwise σ and σ' are the same.

$$\sigma \rightarrow_m^n \sigma' \stackrel{\text{def}}{=} \begin{cases} \exists \sigma''. \sigma \rightarrow_m^{n-1} \sigma'' \wedge \sigma'' \rightarrow_m^1 \sigma' & : n > 0 \\ \sigma = \sigma' & : n = 0 \end{cases}$$

In this transition system, single step transitions are denoted as $\sigma \rightarrow_m \sigma' \stackrel{\text{def}}{=} \sigma \rightarrow_m^1 \sigma'$, we use $\sigma \rightarrow_m^* \sigma'$ for arbitrary long executions, and if $\sigma \rightarrow_m \sigma'$ then σ is in mode m . Moreover, we use $\sigma_0 \rightsquigarrow \sigma_n$ to represent the weak transition relation that holds if there is a finite execution $\sigma_0 \rightarrow \dots \rightarrow \sigma_n$ such that σ_n is in unprivileged mode and all the intermediate states σ_j for $0 < j < n$ are in privileged mode (i.e., the weak transition hides internal states of the hypervisor).

For models having states consisting of components other than memory, the no-exfiltration and no-infiltration properties as we defined above are not sufficient to ensure separation. The behavior of the system in such models does not solely depend on the memory and interference with other components can lead to unexpected misbehavior. Therefore, we try to extend the definition of the security properties to accommodate these additional components. To this end, we give some auxiliary definitions.

Definition (Observation) For a given guest g on the hypervisor, we define the guest observation O_g as all state components that can affect its execution.

Definition (Secure Observation) The remaining part of the state (i.e., the memory of other partitions and some control registers) which are not directly observable by the guest constitute the secure observations O_s of the state.

Definition (Consistent State) We define consistent states as states in which value of components are constrained by a functional invariant. The invariant consists of properties that enforce well-definedness of states; e.g., there is no mapping in a page-table that permits guest accesses to the hypervisor memory. Moreover, Q represents the set of all possible states that satisfy the functional invariant.

The extended no-infiltration guarantees that instructions executed in unprivileged user mode and services executed inside the hypervisor on behalf of the guest maintain equality of the guest observation if the execution starts in consistent states having the same view of the system.

Property 4. Let $\sigma_1 \in Q$ and $\sigma_2 \in Q$ and assume that $O_g(\sigma_1) = O_g(\sigma_2)$, if $\sigma_1 \rightsquigarrow \sigma'_1$ and $\sigma_2 \rightsquigarrow \sigma'_2$ then $O_g(\sigma'_1) = O_g(\sigma'_2)$.

Proving the no-infiltration property over the course of several instructions entails showing the inability of the guest software in changing the critical state components. This prevents a guest from elevating its permissions to access resources beyond its granted privileges, such as the memory allocated for the page-tables or the value of control registers. This can be done by showing that guest transitions preserve the consistency of states and that the secure observation remains constant in all reachable states.

Property 5. Let $\sigma \in Q$, if $\sigma \rightarrow_0 \sigma'$ then $O_s(\sigma) = O_s(\sigma')$ and $\sigma' \in Q$.

Due to the interleaving of the hypervisor and guests executions, one has to check that context switching (i.e., changing processor's execution mode) to the hypervisor is done securely to prevent a guest from gaining privileged access rights. More importantly, it must be ensured that *program counter* cannot be loaded with an address of the guest's choice, and all entry points into privileged mode should be in an exception handler. This guarantees that the guest cannot execute arbitrary code in privileged mode. Likewise, it needs to be ensured that interrupts are correctly masked, and the return address belongs to the interrupted guest and is properly stored. To check these properties, we define a *switching convention* that serves to restore the context for the activated mode and checks that mode switching from unprivileged mode to privileged mode is done securely.

Property 6. Let $\sigma \in Q$ be an arbitrary state in unprivileged mode and σ' be a state in which the hypervisor is active. If $\sigma \rightarrow_0 \sigma'$ then context switching complies with the switching convention.

Additionally, it is also essential to show that execution of hypervisor handlers maintains the functional invariant. Since the hypervisor's transitions can break the

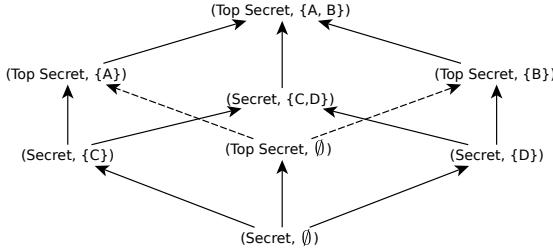


Figure 2.7: A lattice of security labels. Arrows show the intended information channels, moreover, A , B , C , and D are objects in the system.

invariant, we do not check the intermediate privileged steps. However, it must be ensured that the hypervisor yields control to the guest only in a state which satisfies the invariant.

Property 7. *Let $\sigma \in Q$ is the state immediately after switching to the kernel, if $\sigma \rightsquigarrow \sigma'$ then $\sigma' \in Q$.*

Information Flow Security

Information flow analysis, for systems implementing the MILS concept, is the study of controlling the propagation of information among security levels. The primary objective of this study is to rigorously ensure that there are no “illegal” flows of high-criticality data to low-criticality observers. Denning [75] used a lattice of security labels to perform this analysis. In her approach security labels indicate criticality level of objects (e.g., files or program variables) or information receptacles, and the lattice structure represents the *information flow policy* within the system. An example of such a lattice is depicted in Figure 2.7.

The seminal paper by Goguen and Meseguer [92, 91] was first to coin the concept of *noninterference*. They divide the system into a number of *security domains* and information can flow between domains only according to the information flow policy of the system. Goguen-Meseguer noninterference prevents actuators in one domain from learning about the occurrence of certain events in other ones. Pure noninterference, which is also commonly known as no-infiltration, forbids any flow of information among domains. This would ensure that actions performed by one domain cannot influence subsequent outputs seen by another one, and thus the system does not leak confidential data.

Rushby defined the notion of *intransitive noninterference* [181] as a declassification of classical noninterference, which can accommodate the flow of information between domains. In a later work, von Oheimb [216] revisited noninterference defined by Rushby and proposed concepts of *nonleakage* and *noninfluence* for state-based systems. Nonleakage is a confidentiality property prohibiting domains from

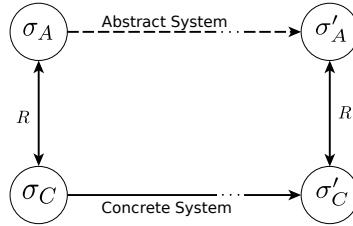


Figure 2.8: Forward simulation.

learning about private data of one another, but it does not forbid them from learning about the occurrence of transitions of other domains. Noninfluence is defined as the combination of nonleakage and Goguen-Meseguer noninterference.

2.2.4 Verification Methodology

We conclude this section by giving an overview of our proof strategy and drawing a connection between tools that we have used for formal verification. Our strategy to analyze the PROSPER kernel, to examine its security and correctness, consists of lifting the main body of our reasoning to a high-level (design) model of the system, which is derived from the real implementation of the kernel. The main purpose of defining such an abstraction is to facilitate formal reasoning. This abstract model, which is also called Top Level Specification (TLS) or *ideal model*, defines the desired behavior of the system and serves as a framework to check the validity of the security properties [177].

The connection between different layers of abstraction relies on *data refinement* [21, 177]. Refinement establishes the correspondence between the high-level (abstract) model and real implementation, and it shows that results of the analysis on the abstract model can be transferred to the system implementation, too. Since verifying properties is in general easier on an abstract model, a refinement-based approach simplifies the verification process.

Back was first to formalize the notion of *stepwise refinement*, as a part of the *refinement calculus* [22, 23]. This approach is later extended by Back and von Wright to support refinement of data representations, called *data refinement* [21]. Data refinement correctness is often validated by establishing either *bisimulation* or *forward simulation* [145], as logical tools to show the behavioral equivalence between two systems.

For the given concrete implementation C and abstract model A , the implementation C refines A , or equivalently A simulates C , if starting at the corresponding initial states, execution of C and A terminates in the corresponding final states (cf. Figure 2.8). The proof of such a refinement is usually done by defining an

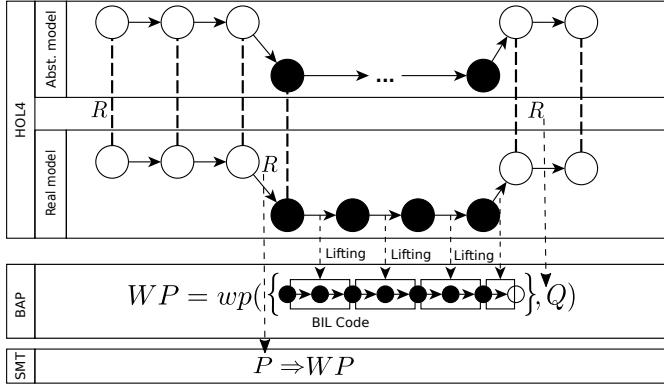


Figure 2.9: Executions of a real machine (middle), and the Top Level Specification (top) and the relations between them. In addition the binary verification methodology (bottom) is depicted. Here, “Lifting” refers to the process of translating the binary code of the kernel to BAP’s IL, P and Q are pre- and post-conditions serving as the contract for a kernel handler, and WP is the weakest precondition computed by taking into account effects of the handler machine code.

abstraction relation $R \subseteq \Sigma_A \times \Sigma_C$ ⁸ between states of the models and showing (by an inductive proof) that R is a simulation relation if A and C are advanced (*step forward*) in parallel. Note that for deterministic systems whose executions depend on the initial state, since for every corresponding initial states there is only one pair of corresponding executions, it is enough to prove the simulation once.

The validity of the simulation relation R between the two models ensures that the abstraction A models all possible behavior of C ; this is called *soundness* of the simulation. However, the inverse relation does not necessarily hold. On the other hand, the relation R is called a bisimulation relation between the two models if, and only if, both the models are simulating each other⁹. This requires showing *completeness* of the relation; i.e., it must be proved that for every execution of A there exists a simulating execution in C .

If the bisimulation relation is proved to hold for all executions of A and C , any security property shown for A also holds on all executions of the implementation C . Additionally, hyperproperties (i.e., properties relating different executions of the same system) proven about A can be transferred to C as well.

Figure 2.9 illustrates our verification strategy. In particular, it depicts the differ-

⁸Establishing the simulation relation R between two models usually entails showing several properties, among which demonstrating *observational equivalence* between the models is an important property in this thesis.

⁹This condition does not necessarily is true for nondeterministic systems. That is, it is possible to have two machines that simulate each other, but are not bisimilar.

ent layers of abstraction, how they are related, and the tools we have used to verify properties at each layer. White circles in Figure 2.9 represent states in unprivileged user mode, and black circles indicate states where the hypervisor is active. In this figure the “Abstract model” represents the top level specification, “Real model” is actual implementation of the system where each transition represents the execution of one binary instruction, and the relation R denotes the refinement relation.

In our approach, the (bi)similarity of unprivileged transitions in two models is established in HOL4. For the privileged transitions, however, proof of the refinement relation R is done using a combination of HOL4 and the BAP tools. Since during the execution of handlers no guest is active, internal hypervisor steps cannot be observed by guests. Moreover, as the hypervisor does not support preemption, the execution of handlers cannot be interrupted. Therefore, we disregard internal states of the handlers and limit the relation R to relate only states where the guests are executing. To show refinement, we use HOL4 to verify that the refinement relation transfers security properties to the Real model and to prove a theorem that transforms the relational reasoning into a set of contracts for the handlers and guarantees that the refinement is established if all contracts are satisfied. Contracts are derived from the hypervisor specification and the definition of the relation R . Then we use BAP to check that the hypervisor code respects the contracts, which are expressed as Hoare triples.

Paper A elaborates more on this pervasive approach that we have adopted to verify the hypervisor. Verification presented in paper B is restricted to proving the correctness of a runtime monitor only in an abstract model of the system. Moreover, paper D shows how properties transferred to an implementation model (using a similar approach as of paper A) can be further transferred down to a model augmented with additional hardware features, namely caches.

2.3 Summary

A hypervisor is a system software that enables secure isolation of critical programs from less trusted (potentially malicious) applications coexisting on the same processor. Hypervisors reduce software portion of the system TCB to a thin layer which is responsible for isolating partitions and handling communication between them. The small codebase of hypervisors minimizes their attack surface and enables the use of rigorous reasoning to prove their security and correctness.

Commonly used properties in verification of hypervisors are data-separation and information flow security. One way of verifying these properties is by constructing an abstract model of the system, proving the properties at this abstract level, and then transferring the verified properties to the system’s machine-code by showing a refinement relation.

When modeling a system, one has to take also into account hardware features such as caches which their effects on the system state can invalidate properties verified in the absence of these features.

Chapter 3

Related Work

In this chapter, we review works on different aspects related to the main contributions of this thesis, namely formal verification of low-level execution platforms (Section 3.1), provably secure supervised execution (Section 3.2), and attacks on isolation (Section 3.3).

Verification of system software is a goal pursued for several decades. Past works on formal verification of operating systems include analysis of Provably Secure Operating System (PSOS) [81] and UCLA Secure Unix [219]. Neumann et al. [81] used the Hierarchical Development Methodology (HDM) [176] to design an operating system with provable security properties. HDM is a software development methodology to facilitate formal verification of the design and implementation. UCLA Secure Unix [219] aimed to formally prove that data-separation is properly enforced by the kernel of a Unix-based OS. Verification of UCLA Secure Unix was based on a refinement proof from a top-level abstraction down to the Pascal code of the kernel, and verification assumed the correctness of underlying hardware. KIT [37], for Kernel of Isolated Tasks, is probably the first fully verified operating system. Software used in this exercise was a small idealized kernel which was proved to implement distributed communicating processes. Bevier [37] showed that properties verified at an abstract model of KIT could be refined to hold also for its real implantation. Among others, the *VFiasco*[113] and Robin [204] projects were also conducted to verify system level software.

3.1 Verification of Security-Kernels

Next, we look at a selection of projects on verification of microkernels, separation kernels, and hypervisors; a rather comprehensive list of formally verified kernels can be found in [235].

Formal verification of the seL4 microkernel [133] is probably the leading work in verification of system software. The functional correctness of seL4 has been verified in Isabelle/HOL [159] by proving a refinement. The refinement relation shows the

correspondence between different layers of abstractions from a high-level abstract specification down to the machine code [190] of seL4. Murray et al. [152, 153] later extended the verification of seL4 by showing its information flow security based on the notions of *nonleakage* and *noninfluence* as introduced in [216]. The verification of seL4 assumes a sequential memory model and ignores leakages via cache timing and storage channels. Nevertheless, the bandwidth of timing channels in seL4 and possible countermeasures were examined, later, by Cock et al. [56].

Heitmeyer et al. [108] proposed a practical approach to security analyze separation kernels for embedded devices. The top-level specification used in that project was a state-machine model of the system which provides a precise description of the required behavior and is proved to enforce separation properly. The information flow security of the kernel is shown by annotating its source-code by Floyd-Hoare style assertions and showing that non-secure information flows cannot occur. Verification is done in the PVS theorem prover using a memory coherent (cacheless) model of the system. However, no machine-checked verification was directly done at the implementation level of the kernel.

The primary verification objective of the Verisoft project [213] was to achieve a pervasive formal analysis of the system from hardware to programs running in unprivileged user mode. Verisoft aimed to dismiss the assumption of compiler and instruction set model correctness. Verisoft-XT [214] continued the verification started under the Verisoft project. Verisoft-XT initially targeted the verification of the Microsoft Hyper-V hypervisor using the VCC tool [57]. For this exercise, guests are modeled as full x64 machines, and caches are not transparent if the same memory location is accessed in cacheable and uncachable mode. The verification of Hyper-V was later dropped to use the developed techniques in verification of an idealized hypervisor for a baby VAMP architecture [11]. They fully automate the verification process using VCC and proved that the virtualization of the memory management subsystem is correct [8]. Verisoft-XT also made contributions to multi-core concurrency and verification under weak memory models [59]. A subproject of Verisoft-XT was to demonstrate the functional correctness of the PikeOS [33] microkernel at its source-code level. The verification is done in VCC by establishing a simulation relation between a top-level abstract model and the real implementation of the system.

mCertiKOS [98] is a hypervisor that uses the hardware virtualization extensions to virtualize the memory subsystem and to enforce isolation of partitions. mCertiKOS runs on a single-core processor, and its functional correctness [98] and noninterference property [65] has been verified based on a sequential memory model within the proof assistant Coq. The follow-up work by Gu et al. [99] extended the CertiKOS hypervisor to run on a multi-core processor. The Contextual functional correctness of the extend kernel also has been verified using the Coq theorem prover. CertiKOS is implemented in ClightX, and they used CompCertX to compile and link kernel programs.

Among other related works, Ironclad [104] is a research project which applied formal methods to verify the system's entire software stacks, including the kernel,

user mode applications, device drivers, and cryptographic libraries. The verification of Ironclad to prove its functional correctness and information flow security has been performed on a cacheless model down to the assembly implantation of the system. Similarly, Barthe et al. [29, 30] applied formal reasoning to show security of an idealized hypervisor, they also included an abstract model of caches in their analysis and demonstrated how the isolation property can be verified when caches are enabled.

To formally verify the INTEGRITY-178B separation kernel [174] the GWV policy [96]¹ was extended to describe the dynamic scheduling of the kernel [103]. This extended policy was also able to capture the flow of information within the system completely. The analysis is done by creating three specifications, namely a functional specification which represents the functional interfaces of the system, a high- and low-level designs that are the semi-formal representation of the system with different levels of details. The correspondence between the specifications was shown by a “code-to-spec” review process. The verification of INTEGRITY-178B to some extent is accomplished using the ACL2 theorem prover.

3.1.1 Trustworthy MMU Virtualization

The memory subsystem is a critical resource for the security of low-level software such as OS kernels and hypervisors. Since devices like MMUs determine the binding of physical memory locations to locations addressable at the application level, circumventing the MMU provides a path for hostile applications to gain illicit access to protected resources. It is therefore of interest to develop methods for the MMU virtualization that enables complete mediation of MMU settings and can protect trusted components. Examples of systems that use virtualization of the memory subsystem to protect security-critical components include KCofi [66] based on the Secure Virtual Architecture (SVA) [69], Overshadow [49], Inktag [112], and Virtual Ghost [67].

KCofi [66], for Kernel Control Flow Integrity, is a system to provide complete Control-Flow Integrity protection for COTS OSs. KCofi relies on SVA [69] to trap sensitive operations such as the execution of instructions that can change MMU configurations. SVA is a compiler-based virtual machine that can interpose between hardware and the operating system. The functional correctness of KCofi has been verified in Coq. The verification covers page-table management, trap handlers, context switching, and signal delivery in KCofi.

Overshadow [49] is a hypervisor to protect confidentiality and integrity of legacy applications from commodity operating systems. Overshadow provides OS with only an encrypted view of application data. This encrypted view prevents the OS’s kernel from performing unauthorized accesses to application data while enabling

¹The GWV security policy allows controlled communication of partitions executing on a separation kernel. GWV restricts effects on memory *segments* in a partition to memory regions that are (i) associated with the current partition and (ii) allowed to interact with the segments, according to the Direct Interaction Allowed (dia) function.

the kernel to manage system resources. Overshadow yields a different view of the memory depending on the context performing a memory operation.

Techniques mostly used to virtualize the memory subsystem are shadow paging, nested paging, and microkernels. The functional correctness of mechanisms based on shadow page-tables has been verified in [8, 11, 107, 138]. XMHF [211] and CertiKOS [98] are examples of verified hypervisors for the x86 architecture that control memory operations through hardware virtualization extensions.

XMHF [211], or eXtensible and Modular Hypervisor Framework, is a formally verified security hypervisor, which relies on hardware extensions to virtualize the memory subsystem and to achieve high performance. The XMHF hypervisor design does not support interrupts. This design choice enabled automated verification of the memory subsystem integrity using model checking techniques.

3.2 Trustworthy Runtime Monitoring

Security kernels [14] or reference monitors are programs used traditionally to enforce access control policies in the kernel of mainstream operating systems. To fulfill the requirements of having an efficient monitoring subsystem (i.e., providing the monitor with a complete view of the system and making it tamper-resistant), the monitor can be implemented as a loadable kernel module. Nevertheless, kernel level solutions [144, 228] are not reliable, as the kernel itself is vulnerable to attacks, such as code injection attacks and kernel rootkits.

An alternative is protecting the monitor by a more privileged layer, such as a hypervisor. The protection can be done through either making inaccessible from the hypervisor, memory of the monitor, e.g., [192], or integrating the monitor into the virtualization layer [189, 128]. The later, however, has the inconvenience of increasing the complexity of the hypervisor itself, invalidating the principle of keeping the TCB as minimal as possible. Secvisor [189, 83] is a formally verified hypervisor based runtime monitor to preserve the guest’s kernel integrity. Secvisor uses the MMU virtualization to protect the kernel memory and to defend itself against attacks. SecVisor was verified using the model checking approach to ensure that only user-approved code can execute in privileged mode of the CPU.

More reliable solutions to implement a secure runtime monitor are protecting the monitor using hardware extensions, e.g., ARM TrustZone [1], or security kernel. Example of softwares which use hardware extensions to protect the monitoring module includes [165, 222]. Lares [165] is a runtime monitor built on a hypervisor. Lares consists of three components: a partition to execute a guest software, a monitoring module deployed in a separate partition, and the hypervisor which creates isolated partitions and supplies the monitor with a complete view of the guest. Hypervision [222] uses isolation at the hardware level to protect the monitoring subsystem and to have full control over the target kernel’s memory management. Hypervision is entirely located within ARM TrustZone and uses hardware features to intercept security-critical events of the software that it protects.

3.3 Attack On Isolation

Creating an isolated, integrity-protected processing environment for security-critical computations is an active research area in platform security. Over the last years, the advent of new technologies [15, 148, 1] has made this long-standing goal (almost) attainable. However, attacks targeting these mechanisms raised concerns about security of these technologies themselves as the system *root of trust*. There is a large body of papers that demonstrate attacks successfully conducted to subvert techniques employed to create trusted execution environments. This section overviews a few works on exploiting vulnerabilities to compromise isolation solutions.

Hypervisor level attacks can be, in general, categorized into two groups: *side-channel attacks* and *hypervisor level malware attacks*. Side-channels, as defined in Chapter 2, are paths that exist accidentally to the otherwise secure flow of data and through which confidential information can escape. Side-channels are usually built using hidden features of system components; among others, memory and caches are extensively used to construct such channels. Jankovic et al. [167] showed how to construct a side-channel between virtual machines on a hypervisor using the *Flush+Reload* attack. Apecechea et al. [122] used *Bernstein's correlation attack* [40] to create a side-channel between partitions executing on Xen and VMware [178] to extract the secret key of the AES cryptographic algorithm. Similarly, in [230] the last level cache (i.e., L3 cache) is used to mount a side-channel attack.

Hypervisor level malware attacks are used mostly to detect the presence of the virtualization layer [140] and to mimic the structure of a hypervisor [130]. The latter, also called *hyperjacking*, installs as a bare-metal hypervisor and moves the victim software inside a partition without being detected. An example of this type of attacks is the *Blue-Pill* root-kit developed by security researcher Joanna Rutkowska.

Hypervisors are not the only victim of attacks on isolation. There is an extensive list of attack techniques that target hardware level solutions such as ARM TrustZone and Intel SGX. TrustZone was first introduced in ARMv6 to create an isolated execution environment. However, it is vulnerability to attacks from unprivileged user space, to execute arbitrary code inside TrustZone's secure world, is shown in [193]. Intel *Software Guard Extensions* is a technology to protect select data from disclosure or modification. Nevertheless, a study by Costan and Devadas [64] and similarly [42, 95] revealed the vulnerability of this technology to cache attacks and software side-channel attacks. It is also acknowledged by Intel [120] that SGX does not provide protection against side-channel attacks including those that constructed through exploiting performance counters.

3.4 Summary

Formal verification of system software has made great strides in recent years. Verifications are mostly done on models that are simpler than current processors. These

verification exercises represent significant investments in development and verification. However, excluding hardware features such as caches, TLBs, and pipelines from the analysis makes properties verified in the absence of these features not reliable in a richer setting.

Taking control of the memory management subsystem, e.g., through virtualizing the MMU, is a widely adopted technique to enforce isolation of components in the system. Many projects used memory isolation to deploy a runtime monitor to control system activities securely.

Chapter 4

Contributions

The main contributions of this thesis can be split into three parts: (i) design and implementation of a security hypervisor, (ii) attacking the isolation guarantees of the hypervisor through constructing a low-noise cache storage side-channel, and (iii) formal verification of the hypervisor down to its machine-code. Our approach to secure the hypervisor is based on taking control of the memory management subsystem by designing an algorithm to virtualize the MMU. Furthermore, we use formal methods to verify the functional correctness and security of the virtualization mechanism.

This thesis consists of four articles, some of which originally published in peer-reviewed journals and conference proceedings. In this chapter, we give a summary of these papers together with a statement of the author's contributions. Some papers are revised to improve presentation and to include proofs of key theorems. In particular, some details in paper B concerning the proof of important theorems have been changed from the original paper.

4.1 Summary of Included Papers

Paper A: Provably secure memory isolation for Linux on ARM.

Originally published as *Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for Linux on ARM. Journal of Computer Security, 24(6):793-837, 2016.*

Content Necessary for a hypervisor to host a general purpose operating system is permitting the guest software to manage its internal memory hierarchy dynamically and to impose its access restrictions. To this end, a mandatory security requirement is taking control of the memory management subsystem by the hypervisor through virtualizing the MMU to mediate all its configurations. The MMU virtualization allows isolating security-critical components from a commodity OS running on the same processor. This enables the guest OS to implement noncritical

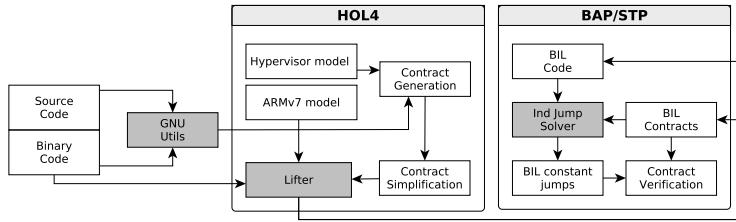


Figure 4.1: Binary verification workflow: *Contract Generation*, generating pre- and post conditions based on the system specification and the refinement relation; *Contract Simplification*, massaging contracts to make them suitable for verification; *Lifter*, lifting handlers machine-code and the generated contracts in HOL4 to BIL; *Ind Jump Solver*, the procedure to resolve indirect jumps in the BIL code; *BIL constant jumps*, BIL fragments without indirect jumps; *Contract Verification* using SMT solver to verify contracts. Here, gray boxes are depicting the tools that have been developed/extended to automate verification.

functionalities while critical parts of the system are adequately protected. In this paper, we present the design, implementation, and verification of a memory virtualization mechanism. The hypervisor targets a common architecture for embedded devices, namely ARMv7 architecture, and it supports execution of Linux without requiring special hardware extensions. Our virtualization approach is based on *direct paging*, which is inspired by the paravirtualization mechanism of Xen [26] and Secure Virtual Architecture [69]. We show that direct paging can be implemented using a compact design that is suitable for formal verification. Moreover, using a refinement-based approach, we prove complete mediation along with memory isolation, and information flow correctness of the virtualization mechanism. Verification is done on a high-level model that augments a real machine state with additional components which represent the hypervisor internal data structure. We also demonstrate how the verified properties at the high-level model of the system can be propagated to the hypervisor’s machine-code.

The binary verification relies on Hoare logic and reduces the high-level (relational) reasoning in HOL4 into verification of some contracts expressed in terms of Hoare triples $\{P\} C \{Q\}$. To validate contracts, we compute the weakest precondition of binary fragments in the initial state to ensure that the execution of the fragment terminates in a state satisfying the postcondition. We then prove that the precondition entails the weakest precondition. The validity of these contracts shows properties verified at high-level models are also valid for the machine-code of the hypervisor.

The binary verification of the hypervisor is automated to a large extent using the BAP tool (cf. Figure 4.1). To use BAP we *lift* the ARMv7 assembly to the intermediate language of BAP. Lifting is done by utilizing a tool developed in HOL4

that generates for each expression a certifying theorem showing the equality of the expression’s BIL fragment and the corresponding predicate in HOL4. Several other tools have also been developed to automate the verification process and to optimize the weakest precondition generation algorithm of BAP. The optimization is needed to reduce the size of generated predicates, which otherwise can grow exponentially due to the number of instructions. Validating contracts by computing the weakest precondition relies on the absence of indirect jumps. To fulfill this requirement, we have implemented a simple iterative procedure that uses STP (an SMT solver) [84] to resolve indirect jumps in the code. Moreover, writing predicates and invariants usually need information on data types together with location, alignment, and size of data structure fields. Since machine-code (and BIL) lacks such information, we developed a set of tools that integrate HOL4 and the GNU Debugger (GDB) [87] to extract the required information from the C source code and the compiled assembly.

In this paper, we also present several applications of the hypervisor. For instance, we show how a runtime monitor can be deployed on the hypervisor to check internal activities of a Linux guest executing in a separate partition.

Statement of Contributions For this paper, I was responsible for adapting the direct paging algorithm to the PROSPER kernel, implementing the memory management subsystem of the hypervisor, porting the Linux kernel on the hypervisor, and developing the new front-end for the BAP tool. The Linux port is done jointly with other coauthors and our colleagues from Swedish Institute of Computer Science (SICS). Further, I together with Roberto Guanciale did all the modeling and verification of the hypervisor. I also contributed to preparing the text.

Paper B: Trustworthy Prevention of Code Injection in Linux on Embedded Devices.

Originally published as *Hind Chfouka, Hamed Nemati, Roberto Guanciale, Mads Dam, and Patrik Ekdahl. Trustworthy prevention of code injection in Linux on embedded devices. In Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I, pages 90-107, 2015.*

Content This paper demonstrate a use-case of isolation provided by the PROSPER hypervisor. In particular, the paper presents the design, implementation, and verification of a VMI-based runtime monitor. The primary goal of the monitor is thwarting code injection attacks in an untrusted guest OS. The monitor is placed in an isolated virtual machine to check internal activities of a Linux guest running in a separate partition. Deploying the monitor inside a dedicated partition has the advantage of decoupling the policy enforcement mechanism from the other hypervisor’s functionalities and keeping the TCB of the system minimal. The security policy enforced by the monitor is $W \oplus X$; i.e., each memory block within the Linux memory space can be either writable or executable but not both at the

same time. This prevents buffer overflow attacks and guarantees that the untrusted guest is not able to directly modify executable memory blocks. The monitor uses a signature-based validation approach to check the authenticity of executables in the Linux memory. For this, the monitor keeps a database of valid signatures for known applications. Whenever Linux tries to run an executable, the monitor intercepts this operation, due to the executable space protection policy, and checks if this executable has a valid signature in the database. If not, the monitor prohibits this operation.

The verification is carried out on an abstract model of the system to show that the monitor guarantees the integrity of the system. The verification is performed using the HOL4 theorem prover and by extending the existing hypervisor model with the formal specification of the monitor.

Statement of Contributions For this paper, I contributed to the design, implementation, and verification of the monitor. The design and implementation are done together with Roberto Guanciale, and I also contributed to the verification of the monitor and writing the paper text.

Paper C: Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures.

Originally published as *Roberto Guanciale, Hamed Nemat, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22–26, 2016, pages 38–55, 2016.*

Content Resource sharing, while inevitable on contemporary hardware platforms, can impose significant challenges to security. First level data-cache is one such shared resource that is transparent to programs running in user mode. However, it influences the system’s behavior in many aspects.

Great as the importance of caches for the system performance is, caches are widely used to construct side-channels. These channels enable unauthorized parties to gain access to confidential data through measuring caches’ effect on the system state. As an instance, cache-timing channels are attack vectors built via monitoring variations in execution time, due to the presence of caches, and used in several cases to extract secret key of cryptographic algorithms.

This paper presents a new attack vector on ARMv7 architecture, which exposes a low-noise cache side-channel. The vector uses virtual aliasing with mismatched cacheability attributes and self-modifying code to build cache “storage” side-channels. We use this attack vector to break the integrity of an ARMv7 hypervisor and its trusted services, such as a runtime monitor which was verified on a cacheless model, as presented in paper B of this thesis. Furthermore, we show how this vector can be used to attack the confidentiality of an AES cryptographic algorithm that was placed in the secure world of platform’s TrustZone.

To counter the cache storage side-channel attack we propose several countermeasures and implemented some of them to show their effectiveness. Additionally, we informally discuss how the formal verification of software which has been previously verified in a memory coherent (cacheless) model can be restored on platforms with enabled caches.

Statement of Contributions Roberto Guanciale and I initially developed the idea of building cache storage side-channels using mismatched cacheability attributes. Then, I investigated more the possibility of constructing such channels and implemented the proof-of-concept attack based on the code provided by Arash Vahidi (at SICS). I was also responsible for developing the integrity attack against the PROSPER kernel and implementing some of the countermeasures inside the hypervisor and the Linux kernel. Moreover, I contributed to discussions on fixing the formal verification of software previously verified in a cacheless model, and writing the paper text.

Paper D: Formal Analysis of Countermeasures against Cache Storage Side Channels

*Hamed Nemati, Roberto Guanciale, Christoph Baumann, and Mads Dam.
Formal Analysis of Countermeasures against Cache Storage Side Channels.*

Content Over the past decades, formal verification has emerged as a powerful tool to improve the trustworthiness of security-critical system software. Verification of these systems, however, are mostly done on models abstracting from low-level platform details such as caches and TLBs. In [100] we have demonstrated several attacks built using caches against a verified hypervisor. Such attacks are facts showing that excluding low-level features from formal analysis makes the verification of a system unreliable and signify the need to develop verification frameworks that can adequately reflect the presence of caches.

We informally discussed in [100] how to restore verified (on a cacheless model) guarantees of software when caches are enabled. In a similar vein, in paper D we formally show how a countermeasure against cache attacks helps (1) restoring the integrity of a software and (2) proving the absence of cache storage side-channels. To ease the burden of verification task, we also try to reuse the previous analysis of the software on the cacheless model, as much as possible.

To carry out the analysis, we define a cacheless model, which is memory coherent by construction, and a cache-aware model. The cache-aware model is a generic ARM core augmented with caches and is sufficiently detailed to reflect the behavior of an untrusted application that can break memory coherence. We show for privileged transitions that integrity and confidentiality can be directly transferred to the cache-aware model if the two models behave equivalently. However, since user level applications are unknown and free to break their coherency, these properties

cannot be transferred without overly restricting applications' transitions. Instead, we revalidated security of unprivileged transitions in the cache-aware model.

Showing behavioral equivalence of the models entails proving that countermeasures are correctly implemented and are able to restore memory coherence. This, in turn, generates some proof obligations that can be imposed on the cacheless model, thus permitting to use existing tools [24, 47, 196] (mostly not available on a cache enabled model) to automate verification to a large extent.

The main feature of our approach is the decomposition of proofs into (i) software dependent (ii) countermeasure dependent and (iii) hardware dependent parts. This helps to verify the countermeasures once and independent of the software executing on the system and makes easier to adopt the verified properties for different software or hardware platforms. We used HOL4 to machine-check our verification strategy to prove kernel integrity and user level properties. However, kernel confidentiality is so far mainly a pen-and-paper proof.

Statement of Contributions Initially, I tried to develop a proof methodology to fix verification of a low-level execution platform on a cache-aware model, and I wrote the initial draft of the paper. Then, all authors contributed to improve the technical development and writing the text.

4.2 Further Publications

In addition to the included papers, the following papers have been produced in part by the author of this thesis:

- Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, Oliver Schwarz: Formal verification of information flow security for a simple arm-based separation kernel. ACM Conference on Computer and Communications Security 2013: 223-234.
- Mads Dam, Roberto Guanciale, Hamed Nemati: Machine code verification of a tiny ARM hypervisor. TrustED@CCS 2013: 3-12.
- Hamed Nemati, Mads Dam, Roberto Guanciale, Viktor Do, Arash Vahidi: Trustworthy Memory Isolation of Linux on Embedded Devices. TRUST 2015: 125-142.
- Hamed Nemati, Roberto Guanciale, Mads Dam: Trustworthy Virtualization of the ARMv7 Memory Subsystem. SOFSEM 2015: 578-589.

Chapter 5

Conclusions

In this thesis, we explore the design, implementation, and verification of a security hypervisor as an enabler for isolation between system components. The provided isolation allows building mixed-criticality systems which consolidate critical functionalities with convenience features on the same processor. The hypervisor reduces the size of the system's TCB, and its small codebase permitted us to apply formal methods to show that the virtualization layer is functionally correct and behaves as specified. Further proofs are also used to ensure that the hypervisor preserves the system integrity and confidentiality, guaranteeing that separated components are not able to affect each other.

5.1 Contribution

The contributions of this thesis are summarised as follows. In paper A, we use direct paging to implement a provably secure virtualized memory management subsystem for embedded devices. We formally verify on a high-level model of the system that our approach is functionally correct, and it properly isolates partitions. By proving a refinement theorem, we then show how these properties can be (semi-) automatically propagated to the binary code of the hypervisor. A use-case scenario for isolation obtained using memory virtualization is presented in paper B. In that paper, we show how a trustworthy runtime monitor can be securely deployed in a partition on the hypervisor to prevent code injection attacks within a fellow partition, which hosts a COTS operating system. The monitor inherits the security properties verified for the hypervisor, and we proved that it enforces the executable space protection policy correctly.

In the related works chapter, we have seen that verification of system software is mostly done on models that are far simpler than contemporary processors. As a result, there are potential attack vectors that are not uncovered by formal analysis. Paper C presents one such attack vectors constructed by measuring caches effects. The vector enables an adversary to breach isolation guarantees of a verified system

on sequential memory model when is deployed on richer platforms with enabled caches. To restore integrity and confidentiality, we provide several countermeasures. Further, in paper D, we propose a new verification methodology to repair verification of the system by including data-caches in the statement of the top-level security properties.

5.2 Concluding Remarks and Future Work

Whether the results presented in this work will contribute to any improvement in a system’s overall security posture depends on many factors. In particular, it depends on characteristics of hardware, software executing on the platform, and capabilities of adversaries. In this thesis, we tried to address only a subset of problems related to platform security and soundness of our results are constrained by the hypervisor design, model of the processor in HOL4, and hardware platforms that we have used as our testbeds.

For instance, the model of the memory management subsystem which we have used in our formal analysis is manually derived from ARM Architecture Reference Manuals [17, 63, 2]. ARM specifications are mostly written in a combination of natural language and pseudocode and leave some aspects of the processor behavior (e.g., effects of executing some system-level instructions) under-specified. This informal and imprecise description makes developing formal models from such specifications laborious and error-prone and potentially undermines results achieved based on these models. ARM admits this problem and declares the significance of having more precise and trustworthy specifications as a motivation behind their recently published machine-readable and executable specifications [173]. A future direction for our work is adopting models produced directly from such machine-readable specifications, which also makes easier to keep up-to-date the formal model of the system as hardware changes.

Moreover, the complexity of today’s hardware and system software is such that a verification approach allowing reuse of models and proofs as new features are added is essential for formal verification of low-level execution platforms to be economically sustainable. The HASPOC project [32] showed how to adopt a compositional verification strategy to attain this goal. They modeled system using several automata, each with a separate specification, which interact via message passing. Such compositional approach allows delaying the implementation of detailed models for some components while making feasible to verify high-level security properties of the system like integrity and confidentiality. A further advantage of decomposing the model of a system into separate components is that guarantees on constant parts can be reused when other parts change.

There are a few other issues that should be addressed before our hypervisor can get adopted in real-world scenarios. For example, the current hypervisor does not allow explicit communication between partitions. Enabling communication, however, makes formal verification challenging. The reason for this is that permit-

ting the flow of information between system components makes it infeasible to use classical noninterference properties like *observational determinism* [150] to show system confidentiality. In such a setting, proving isolation requires a careful analysis to ensure partitions cannot infer anything more than what is allowed by the information flow policy about one another's internal states, e.g., the secret key of a cryptographic algorithm stored in partition memory.

Finally, in this thesis, we restrict our experiments to software running on a single-core processor connected to first-level caches, namely L1- data and instruction caches. Further investigations would be needed to understand the security impact of other components such second-level caches, pipelines, branch prediction unit, TLBs, and enabling multi-core processing on the platform, some of which are left for future work and discussed in more details in respective papers.

Part II

Included Papers

Paper A

Provably secure memory isolation for Linux on ARM

Roberto Guanciale, Hamed Nemati, Mads Dam, Christoph Baumann

Abstract

The isolation of security-critical components from an untrusted OS allows to both protect applications and to harden the OS itself. Virtualization of the memory subsystem is a key component to provide such isolation. We present the design, implementation and verification of a memory virtualization platform for ARMv7-A processors. The design is based on direct paging, an MMU virtualization mechanism previously introduced by Xen. It is shown that this mechanism can be implemented using a compact design, suitable for formal verification down to a low level of abstraction, without penalizing system performance. The verification is performed using the HOL4 theorem prover and uses a detailed model of the processor. We prove memory isolation along with information flow security for an abstract top-level model of the virtualization mechanism. The abstract model is refined down to a transition system closely resembling a C implementation. Additionally, it is demonstrated how the gap between the low-level abstraction and the binary level-can be filled, using tools that check Hoare contracts. The virtualization mechanism is demonstrated on real hardware via a hypervisor hosting Linux and supporting a tamper-proof run-time monitor that provably prevents code injection in the Linux guest.

A.1 Introduction

A basic security requirement for systems that allow software to execute at different levels of security is memory isolation: The ability to store a secret or to enforce data integrity within a designated part of memory and prevent the contents of this memory to be affected by, or leak to, parts of the system that are not authorised to

access it. Without the usage of special hardware, trustworthy memory isolation is dependent on the OS kernel being correctly implemented. However, given the size and complexity of modern OSs, the vision of comprehensive and formal verification of commodity OSs is as distant as ever.

An alternative to verifying the entire OS is to delegate critical functionality to special low-level execution platforms such as hypervisors, separation kernels, or microkernels. Such an approach has some significant advantages. First, the size and complexity of the execution platform can be made much smaller, potentially opening up for rigorous verification. The literature has many recent examples of this, in seL4 [133], Microsoft’s Hyper-V project [138], Green Hills’ CC certified INTEGRITY-178B separation kernel [174], and the Singularity [116] microkernel. Second, the platform can be opened up to public scrutiny and certification, independent of application stacks.

Virtualization-like mechanisms can also be used to support various forms of application hardening against untrusted OSs. Examples of this include KCoFi [66] based on the Secure Virtual Architecture (SVA) [69], Overshadow [49], Inktag [112], and Virtual Ghost [67]. All these examples rely crucially on memory isolation to provide the required security guarantees, typically by virtualizing the memory management unit (MMU) hardware. MMU virtualization, however, can be exceedingly tricky to get right, motivating the use of formal methods for its verification.

In this paper we present an MMU virtualization API for the ARMv7-A processor family and its formal verification down to the binary level. A distinguishing feature of our design is the use of direct paging, a virtualization mechanism introduced by Xen [26] and used later with some variations by the SVA. In direct paging, page tables are kept in guest memory and allowed to be read and directly manipulated by the untrusted guest OS (when they are not in active use by the MMU). Xen demonstrated that this approach has better performance than other software virtualization approaches (e.g. shadow page tables) on the x86 architecture. Moreover, since direct paging does not require shadow data structures, this approach has small memory overhead. The engineering challenge inherent to this project is to design a minimal API that (i) is sufficiently expressive to host a paravirtualized Linux, (ii) introduces an acceptable overhead and (iii) whose implementation is sufficiently small to be subject to pervasive verification for a commodity CPU architecture such as ARMv7.

The security objective is to allow an untrusted guest system to operate freely, invoking the hypervisor at will, without being able to access memory or processor resources for which the guest has not received static permission. In this paper we describe the design, implementation, and evaluation of our memory virtualization API, and the formal verification of its security properties. The verification is performed using a formal model of the ARMv7 architecture [82], implemented in the HOL4 interactive theorem prover.

The proof strategy is to establish a bisimilarity between the hypervisor executing on a formal model of the ARMv7 instruction set architecture and the top level specification (TLS). The TLS describes the desired behaviour of the system con-

sisting of handlers implementing the virtualization mechanism and the behaviour of machine instructions executed by the untrusted guest. The specification of the MMU virtualization API involves an abstract model state that is not represented in memory and thus by design invulnerable to direct guest access. Due to the direct paging approach, however, the page tables that control the MMU are residing in guest memory and need to be modelled explicitly. Hence, it is no longer self-evident that the desired memory isolation properties, no-exfiltration and no-infiltration in the terminology of [108], hold for guests in the TLS, and an important and novel part of the verification is therefore to formally validate that these properties indeed hold.

To keep the TLS as simple and abstract as possible, the TLS addresses page tables directly using their physical addresses. A real implementation cannot do this, but must use virtual addresses instead, in addition to managing its internal data structures. To this end an implementation model is introduced, which uses virtual addresses instead of physical ones and stores the abstract model state explicitly in memory. This provides a very low-level C-like model of handler execution, directly reflecting all algorithmic features of the memory subsystem virtualization implemented by the binary code of the handlers, on the real ARMv7 state, as represented by the HOL4 model. We exhibit a refinement from the TLS to the implementation model, prove its correctness, and show, as a corollary, that the memory isolation properties proved for the TLS transfer to the implementation model. This constitutes the second part of the verification.

The next step is to fill the gap between the verification of this low-level abstraction and the binary level. To accomplish this an additional refinement must be established. Using the same approach as [72], we demonstrate how this can be achieved using a combination of theorem proving and tools that check contracts for binary code. The machine code verification is then in charge of establishing that the hypervisor code fragments respect these contracts, expressed as Hoare triples. Pre and post conditions are generated semi-automatically starting from the specification of the low-level abstraction and the refinement relation. They are then transferred to the binary analysis tool BAP [47], which is used to verify the hypervisor handlers at the assembly level. Several tools have been developed to support this task, including a lifter that transforms ARM code to the machine independent language that can be analysed by BAP and a procedure to resolve indirect jumps. The binary verification of the hypervisor has not been completed yet. However, we demonstrate the methodology outlined above by applying it to prove correctness of the binary code of one of the API calls. The scalability of the approach has been shown in [71], where it was used to verify the binary code of a complete separation kernel.

An alternative approach would be to focus the code verification at the C level. First, such an approach does not directly give assurances at the ISA level, which is our objective. This can be partly addressed by a certifying compiler such as CompCert[139]. However, system level code is currently not supported by such compilers. Moreover, this type of code is prone to break the standard C-semantics,

for example by reconfiguring the MMU and changing the virtual memory mapping of the program under verification as is the case here.

The verification highlighted three classes of bugs in the initial design of the virtualization mechanism:

1. Arithmetic overflows, bit field and offset mismatches, and signed operators where the unsigned ones were needed.
2. Missing checks of self referencing page tables.
3. Approval of guest requests that cause unpredictable behaviours of the ARMv7 MMU.

Moreover, the verification of the implementation model identified additional bugs exploitable by requesting the validation of physical blocks residing outside the guest memory. This last class of bugs was identified because the implementation model takes into account the virtual memory mapping used by the handlers. Finally, the binary code verification identified a buffer overflow.

We report on a port of Linux kernel 2.6.34 and demonstrate the prototype implementation of a hypervisor for which the core component is the verified MMU virtualization API. The complete hypervisor augments the memory virtualization API by handlers that route aborts and interrupts inside Linux. Experiments demonstrate that the hypervisor can run with reasonable performance on real hardware (Beagleboard-xM based on the Cortex-A8 CPU). Furthermore an application scenario is demonstrated based on a trusted run-time monitor. The monitor executes alongside the untrusted Linux system, enforces the W \oplus X policy (no memory area can be writable and executable simultaneously) and uses code signing to prevent binary code injection in the untrusted system.

A.1.1 Scope and limitations

The binary verification of the hypervisor has not been completed yet. However, we demonstrate the methodology outlined above by applying it to prove correctness of the binary code of one of the API calls. The scalability of the approach has been shown in [71], where it was used to verify the binary code of a complete separation kernel. In Section A.9.5 we comment on the tasks that are not automated and need to be manually accomplished to complete the verification.

A.2 Related Work

The size and complexity of commodity OSs make them susceptible to attacks that can bypass their security mechanisms, as demonstrated in e.g. [206, 124]. The ability to isolate security-critical components from an untrusted OS allows non critical parts of a system to be implemented while the critical software remains adequately

protected. This isolation can be used both to protect applications from an untrusted OS as well as to protect the OS itself from internal threats. For example, KCoFI [66] uses Secure Virtual Architecture [69] to isolate the OS from a run-time checker. The checker instruments the OS and monitors its activities to guarantee the control-flow integrity of the OS itself. Related examples are application hardening frameworks such as Overshadow [49], Inktag [112], and Virtual Ghost [67]. In all these cases some form of virtualization of the MMU hardware is a critical component to provide the required isolation guarantees.

Shadow page tables (SPT) is a common approach to MMU virtualization. The virtualization layer maintains a shadow copy of page tables created and maintained by the guest OS. The MMU uses only the shadow pages, which are updated after the virtualization layer validates the OS changes. The Hyper-V hypervisor which uses shadow pages on x86, has been formally verified using the semi automated VCC tool [138]. Related work [11, 164] uses shadow page tables to provide full virtualization, including virtual memory, for “baby VAMP”, a simplified MIPS, using VCC. This work, along with later work [9] on TLB virtualization for an abstract mode of x64, has been verified using Wolfgang Paul’s VCC-based simulation framework [58]. Also, the OKL4-microvisor uses shadow paging to virtualize the memory subsystem [107]. However, this hypervisor has not been verified.

Some modern CPUs provide native hardware support for virtualization. The ARM Virtualization Extensions [5] augment the CPU with a new execution mode and provide a two stage address translation. These features greatly reduce the complexity of the virtualization layer [210]. XHMF [211] and CertiKOS [98] are examples of verified hypervisors for the x86 architecture that control memory operations of guests . using virtualization extensions. The availability of hardware virtualization extensions, however, does not make software based solutions obsolete. For example, the recent Cortex-A5 (used in feature-phones) and the legacy ARM11 cores (used in home network appliances and the 2014 “New Nintendo 3DS”) do not make use of such extensions. Today, the Internet of Things (IoT) and wearable computing are dominated by microcontrollers (e.g. Cortex-M). As the recent Intel Quark demonstrates, the necessity of executing legacy stacks (e.g. Linux) is pushing towards equipping these microcontrollers with an MMU. Quark and the upcoming ARMv8-R both support an MMU and lack two stage page-tables. Generally, there is no universal sweet spot that reconciles the demands for low cost, low power consumption and rich hardware features. For instance, solutions based on FPGAs and soft-cores such as LEON can benefit from software based virtualization by freeing gates not used for virtualization extensions to be used for application specific logic (e.g. digital signal processing, software-defined radio, cryptography).

A virtualization layer provides to the guest OS an interface similar to the underlying hardware. An alternative approach is to execute the commodity OS as a partition of a microkernel, by mapping the OS threads directly to the microkernel threads, thus delegating completely the process management functionality from the hosted OSes to the microkernel (e.g. L⁴Linux). This generally involves an invasive and error-prone OS adaptation process, however. The formal verification

of seL4 [133] demonstrated that a detailed analysis of the security properties of a complete microkernel is possible even at the machine code level [190]. Similarly, the Ironclad Apps framework [104] hosts security services in a remote operating system. Its functional correctness and information flow properties are verified on the assembly level.

In order to achieve trustworthy isolation between partitions, more light-weight solutions can also be employed, namely formally verified separation kernels [174, 71, 33] and Software Fault Isolation (SFI) [218, 233]. The latter has the advantage over the former in that it is a software-only approach, not relying on common hardware components such as MMU and memory protection units (MPU). Nevertheless, both mechanisms are generally not equipped with the functionality needed to host a commodity OS. Conversely, formally verified processor architectures specifically designed with a focus on logical partitioning [225] and information flow control [19] can be used to achieve isolation.

A.2.1 Contributions

We present a platform to virtualize the memory subsystem of a real commodity CPU architecture: The ARMv7-A. The virtualization platform is based on direct paging, a virtualization approach inspired by the paravirtualization mechanism of Xen [26] and Secure Virtual Architecture [69]. The design of the platform is sufficiently slim to enable its formal verification without penalizing the system performance. The verification is performed down to a detailed model of the architecture, including a detailed model of the ARMv7 MMU. This enables our threat model to consist of an arbitrary guest that can execute any ARMv7 instruction in user mode. We prove complete mediation of the MMU configurations, memory isolation of the hosted components, and information flow correctness. Additionally, we present our methodology for the binary verification of hypervisor code and report on first results. So far, one handler has been verified on the binary level. Completing the binary verification for all handlers is work in progress. The viability of the platform is demonstrated via a prototype hypervisor that is capable of hosting a Linux system while provably isolating it from other services. The hypervisor supports BeagleBoard-xM (a development board based on ARM Cortex-A8) and is used to benchmark the platform on real hardware. As the main application it is shown how the virtualization mechanism can be used to support a tamper-proof run-time monitor that prevents code injection in an untrusted Linux guest.

A.3 Verification Approach

In Figure A.1 we give an overview of the entire verification flow presented in this paper. In particular it depicts the different layers of modelling, how they are related, and the tools used. This is discussed in more detail in Section A.3.6.

Our MMU virtualization API is designed for paravirtualization and targets a commodity CPU (ARMv7-A). In such a scenario, the hosting CPU must provide

two levels of execution: privileged and unprivileged. The hypervisor is the only software component that is executed at the privileged level; at this level the software has complete control of the underlying hardware. All other software components (including operating system kernels, user processes, etc.) are executed in unprivileged mode; direct accesses to the sensitive resources must be prevented and all transitions to privileged mode are controlled through the use of exceptions and interrupts.

In addition to the MMU virtualization API itself, as part of the hypervisor, the system is intended to support two types of clients:

- An untrusted commodity OS guest (Linux) running non-critical software (e.g. GUI, browser, server, games).
- A set of trusted services such as controllers that drive physical actuators, run-time monitors, sensor drivers, or cryptographic services.

An example computation of such system is shown in the row labelled “Real model” of Figure A.1. White circles represent states in unprivileged execution level where the untrusted guest (either its kernel or one of its user processes) are running. Gray circles represent unprivileged states where one of the trusted services are in control. Finally, black circles represent states in privileged level where the hypervisor is active. Transitions between two unprivileged states (e.g. 1 → 2) do not cause any exceptions. The transition between the states 2 and 3 is caused by an exception, for example the execution of a software interrupt. Finally, transitions from privileged to unprivileged levels (e.g. 6 → 7) are caused by instructions that explicitly change the execution level.

A.3.1 Attack Model

Due to the size and complexity of a complete Linux system, a realistic adversary model must consider the Linux partition compromised. For this reason, the attacker is an untrusted paravirtualized Linux kernel and its user processes, that maliciously or due to an error may attempt to gain access to resources outside the guest partition. Thus, the attacker is free to execute any CPU instruction in unprivileged mode; it is initially not able to directly access the coprocessor registers, and all attacker memory accesses are initially mediated by the MMU. However, by exploiting possible flaws in the hypervisor the attacker may during the course of a computation gain such access to the MMU configuration, something our security proof shows is in fact not possible. In this work, we assume absence of external mechanisms that can directly modify the internal state of the machine (e.g. external devices or physical tampering). The analysis of temporal partitioning properties (e.g. timing channels as investigated in [55]) is also deliberately left out of this work.

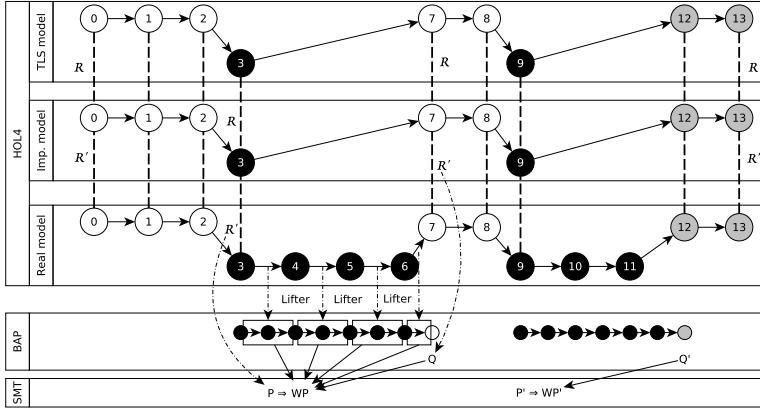


Figure A.1: Executions of a real machine (middle), the implementation model (above), and the Top Level Specification (top) and the relations between them. In addition the dependencies of the binary verification methodology (bottom) are depicted.

A.3.2 Security Goals

The verification must demonstrate that the low level platform does not allow undesired interference between guest and sensitive resources. That is:

1. The hypervisor must play the role of a security monitor of the MMU settings. If *complete mediation* of the MMU settings is violated, then an attacker may bypass the hypervisor policies and compromise the security of the entire system. We show this by proving that neither the untrusted guest nor the trusted services can directly change the MMU configuration.
2. Executions of an arbitrary guest cannot affect the “trusted world”, i.e. the parts of the state the guest is not allowed to modify, such as memory of trusted services, system level registers and status flags, and the hypervisor state. This is an integrity property, similar to the no-exfiltration property of [108].
3. Absence of information flow from the trusted world to the guest, i.e. confidentiality, similar to no-infiltration of [108].

Note that these properties, as in [108], are qualitatively different: The integrity property is a single-trace property, and concerns the inability of the guest to directly write some other state variables. Since it is under guest control when and how to invoke the virtualization API, there are plenty of indirect communication channels connecting guests to the hypervisor. For instance, a guest decision to allocate or deallocate a page table affects large parts of the hypervisor state, without ever

directly writing to any internal hypervisor state variable. Enforcing this is in a sense the very purpose of the hypervisor. On the other hand, the only desired effects of hypervisor actions should be to allocate/deallocate, map, remap, and unmap virtual memory resources, leaving any other observation a guest may make unaffected, thus preventing the guest from extracting information from inaccessible resources even indirectly. This is essentially a two-trace information flow property, needed to break guest-to-guest (or guest-to-service) information channels in much the same way as intransitive noninterference is used in [153] to break guest-to-guest channels passing through the scheduler in seL4.

In this work we establish these properties via successive refinements that add more details (that in turn can highlight different misbehaviour of the system) to the virtualization API, starting from an abstract model refining down to the binary code of the low level execution platform. We first demonstrate that the intended security property holds for the most abstract model. At each refinement, the proof consists of (i) identifying a relation that is strong enough to transfer the security property from the higher abstract model to the more real one (we call this a *candidate relation*) and (ii) demonstrating that the candidate relation actually satisfies the properties required from a refinement relation. For the first task it turns out that one needs a bisimulation relation in order to transfer higher-order information flow properties like confidentiality. The latter task is reduced to subsidiary properties, which have natural correspondences in previous kernel verification literature [108, 174]:

- A malicious guest cannot violate isolation while it is executing.
- Executions of the abstract vs the more real model preserve the candidate bisimulation relation.

These two tasks are qualitatively different. The former task, due to our use of memory protection, is really a noninterference-like property of the hosting architecture rather than a property of the hypervisor. This property must hold independently of the hosted guest, which is unknown at verification time since the attacker can take complete control of the untrusted Linux. By contrast, the latter task consists in verifying at increasing levels of detail the functional correctness of the individual handlers.

A.3.3 Top Level Specification

The first verification task focuses on establishing correctness of the design of the virtualization API. With this goal, in Section A.6.1 we specify the desired behaviour of the virtualization API as a transition system, called the *Top Level Specification* (TLS). This specification models unprivileged execution of an arbitrary guest system on top of a CPU with MMU support, alternating with abstract handler events. These events model invocations of the hypervisor handlers as atomic transformations operating on an abstract machine state. Abstract states are real CPU states extended by auxiliary (model) data that reflect the internal state of the hypervisor.

We refer to this auxiliary data as the abstract hypervisor state. Handler events represent the execution of several instructions at privileged level, in response to exceptions or interrupts. Modelling handler effects as atomic state transformations is possible, since the hypervisor is non-preemptive, i.e. nested exceptions/interrupts are ruled out by the implementation.

Since in direct paging the guest systems can directly manipulate inactive page tables, the TLS needs to explicitly model page tables in memory. This contrasts simpler models such as the one presented in [71] where the hypervisor state was represented in the TLS using abstract model variables only. For this reason, establishing complete mediation, integrity, and confidentiality for the TLS is far from trivial.

A.3.4 Implementation Model

Extending the security properties to an actual implementation, however, requires additional work, for the following reasons:

- The TLS uses auxiliary data structures (the abstract hypervisor state) that are not stored inside the system memory.
- The TLS accesses the memory directly using physical addresses.

As is common practice, the virtualization code executes under the same address translation as the guest (but with different access permissions), in order to reduce the number of context switches required. For this approach it is critical to verify that all low-level operations performed by the hypervisor correctly implement the TLS specification; these operations include reads and updates of the page tables, and reads and updates of the hypervisor data structures. To show implementation soundness we exhibit a refinement property relating TLS states with states of the implementation. The refinement relation is proven to be preserved by all atomic hypervisor operations; reads and updates of the page tables, reads and updates of the hypervisor data structures. In particular it is established that these virtual memory operations access the correct physical addresses and never produce any data abort exceptions. Moreover, it is shown that the refinement relation directly transfers both the integrity properties and the information flow properties of the TLS to the implementation level.

A.3.5 Binary Verification

The last verification step consists in filling the gap between the implementation and the binary code executed on the actual hardware. This requires to exhibit a refinement relation between the implementation model and the real model of the system (i.e. where each transition represents the execution of one binary instruction).

Intuitively, internal hypervisor steps cannot be observed by the guests, since during the execution of the handler no guest is active. Moreover, as the hypervisor

does not support preemption, then the execution of handlers cannot be interrupted. These facts permit to disregard internal states of the handlers and limit the refinement to relate only states where the guests are executing.

Thus, the binary verification can be accomplished in three steps: (i) verification that the refinement relation directly transfers the isolation properties to the real model, (ii) verification of a top level theorem that transforms the relational reasoning into a set of contracts for the handlers and guarantees that the refinement is established if all contracts are satisfied, and (iii) verification of the machine code. The last step establishes if the hypervisor code fragments respect the contracts, expressed as Hoare triples $\{P\}C\{Q\}$, where P and Q are the pre/post conditions of the assembly fragment C .

A.3.6 Proof Engineering

We use Figure A.1 and Table A.1 to summarise the models, theorems and tools that are described in the following sections. We use three transition systems; the TLS (Section A.6.1), the Implementation Model (Section A.6.2) and the ARMv7 model (Section A.4). These transition systems have been defined in the HOL4 theorem prover and differ in the level of abstraction they use to represent the hypervisor behaviour. The three transition systems model guest behaviour identically (e.g. transitions $0 \rightarrow 1$); these transitions obey the access privileges computed by the MMU and satisfy properties 8 and 9 of Section A.4. These properties have been verified for a simplified MMU model in [127].

We use HOL4 to verify that the security properties hold for the TLS (Theorems A.6.1, A.6.2, A.6.3, A.6.4 and A.6.5 of Section A.6.1). The reasoning used to implement the proofs in the interactive theorem prover is summarised in Section A.7.

The refinement (\mathcal{R}) between the TLS and the implementation model is verified in HOL4 (Theorem A.6.6 of Section A.6.2). We also use HOL4 to prove that the refinement transfers the security properties of the TLS to the implementation model (Corollary 1).

The refinement (\mathcal{R}') between the implementation model and the real model is formally defined in HOL4, allowing us to prove that the refinement transfers the security properties to the ARMv7 model (Corollary 2).

The verification of the refinement (Theorem A.6.7 of Section A.6.3) is only partial: we demonstrate the verification of the binary code of the hypervisor only for a part of the code-base and we rely on some assumptions in order to fill the semantic gap between HOL4 and the external tools. We prove Theorem A.6.7 for non-privileged transitions in HOL4 (i.e. transitions not involving the hypervisor code such as $1 \rightarrow 2$ and $12 \rightarrow 13$).

For the hypervisor code, we show that the task can be partially automated by means of external tools. For this purpose we use the HOL4 model of ARMv7 to transform the binary code of the hypervisor (e.g. the code executed between states 3 and 7 in the real model) to the input language of BAP (represented in the figure

by the arrow labelled ‘‘Lifter’’). The usage of HOL4 for this task allows us to reduce the assumptions needed to fill the gap between the HOL4 ARMv7 model and BAP, as described in Section A.9. The methodology to complete the verification is the following: given a hypervisor handler whose code has been translated to the BAP code C , we use a HOL4 certifying procedure that generates a contract $\{P\}C\{Q\}$ starting from the hypervisor implementation model and the refinement relation. The certifying procedure yields a HOL4 theorem stating that the refinement relation \mathcal{R}' is preserved if the hypervisor handler C establishes the postcondition Q starting from the precondition P . We use BAP to compute the weakest precondition WP of the postcondition Q and the code C and a finally an SMT solver checks that the weakest precondition is entailed by the precondition.

A.4 The ARMv7 CPU

ARMv7 is the currently dominant processor architecture in embedded devices. Our verification relies on the HOL4 model of ARM developed at Cambridge [82]. The use of a theorem prover allows the verification goals to be stated in a manner which is faithful to the intuition, without resorting to approximations and abstractions that would be needed when using a fully automated tool such as a model checker. Furthermore, basing the verification on the Cambridge ARM model lends high trustworthiness to the exercise: The Cambridge model is well-tested and phrased in a manner that retains a high resemblance to the pseudocode used by ARM in the architecture reference manual [17]. The Cambridge model has been extended by ourselves to include MMU functionality. The resulting model gives a highly detailed account of the ISA level instruction semantics at the different privilege levels, including relevant MMU coprocessor effects. It must be noted that the Cambridge ARM model assumes linearizable memory, and so can be used out of the box only for processor and hypervisor implementations that satisfy this property, for instance through adequate cache flushing as discussed in Section A.5.5.

We outline the HOL4 ARMv7 model in sufficient detail to make the formal results presented later understandable. An ARMv7 machine state is a record

$$\sigma = \langle uregs, bregs, coregs, mem \rangle \in \Sigma ,$$

where $uregs$, $bregs$, $coregs$, and mem , respectively, represent the user registers, banked registers (used for handling exceptions), coprocessors, and memory. The function $mode(\sigma)$ returns the current privilege execution mode in the state σ , which can be either $PL0$ (unprivileged or user mode, used by the guest) or $PL1$ (privileged mode, used by the hypervisor). The memory is the function $mem \in 2^{32} \rightarrow 2^8$. The coprocessor registers $coregs$ control the MMU.

System behaviour is modelled by the state transition relation $\rightarrow_{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, where a transition is performed by the execution of an ARM instruction. Unprivileged transitions ($\sigma \rightarrow_{PL0} \sigma'$) start from and end in states that are in unprivileged execution mode (i.e. $mode(\sigma) = mode(\sigma') = PL0$). All the other

Artefact	Description	HOL4	BAP	In
TLS	Model of the abstract design of the hypervisor + attacker (guest)	o		[157]
Implementation model	Low level model of the hypervisor + attacker	o		[72]
ARM model	Real model of the system	o		[82, 157]
Properties 8 and 9	Properties of the ARM instruction set (here only assumed)	o		[127]
Properties of the TLS				
Theorem A.6.1	Verification of the functional invariant	o		[157]
Theorem A.6.2	Verification of MMU integrity	o		[157]
Theorem A.6.3	Verification of no context switch	o		[157]
Theorems A.6.4 and A.6.5	Verification of No exfiltration + No infiltration = isolation	o		[157]
Properties of the Implementation model				
Theorem A.6.6	Verification of Refinement	o		[72]
Corollary 1	Verification of MMU integrity + No exfiltration + No infiltration	o		[72]
Properties of the real model				
Theorem A.6.7	Refinement. For non-privileged transitions proved in HOL4. One of the API function proved using BAP	o	o	Here
Corollary 2	Verification of MMU integrity + No exfiltration + No infiltration	o		Here
Miscellaneous				
Lifter	Translation of ARMv7 binary to BIL	o	o	[72, 71]
Certifying procedure	Generates a contract starting from the model of one of the API function and the refinement relation	o		Here
Indirect jump solver	Computes all possible target of indirect jumps for a BIL loop free program. Here extended and re-implemented as BAP extension		o	[72]

Table A.1: List and first appearance of models, theorems and tools.

transitions ($\sigma \rightarrow_{PL1} \sigma'$) involve at least one state in privileged level. The raising of an exception is modelled by a transition that enables the level $PL1$. An exception can be raised because: (i) a software interrupt (SWI) is executed, (ii) the current instruction is undefined, (iii) a memory access is attempted that is disallowed by the MMU, or (iv) an hardware interrupt is received. Whenever an exception occurs, the CPU disables the interrupts and jumps to a predefined address in the vector table to transfer control to the corresponding exception handler.

The ARMv7 MMU uses a two level translation scheme. The first level (L1) consists of a 4096 entry table that divides up to 4GB of memory into 1MB sections. These sections can either point to an equally large region of physical memory or to a level 2 (L2) page table with 256 entries that maps the 1MB section into 4 KB physical pages. MMU behaviour is modelled by the function $mmu(\sigma, pl, va, req)$, which takes a state σ , a privilege level, a virtual address va and an access request $req \in \{rd, wt, ex\}$ (representing read, write and execute accesses) and yields $pa \in 2^{32} \cup \{\perp\}$, where pa is the translated physical address or an access denied. The ARMv7 documentation describes the possibility of unpredictable behaviour due to erroneous setup of the MMU through coprocessor registers and page tables. In this work the hypervisor completely mediates the MMU configuration and aims to rule out this kind of behaviour.

In the ARM architecture *domains* provide a discretionary access control mechanism. This mechanism is orthogonal to the one provided by CPU execution modes. There are sixteen domains, each activated independently in one of the coprocessor registers *coregs*. The page tables map each virtual page/section to one of the domains and the MMU forbids accesses to a page/section if the corresponding domain is not active.

The state transition relation queries the MMU whenever a virtual address is accessed, and raises an exception if the requested access mode is not allowed. To describe the security properties guaranteed by an ARMv7 CPU we introduce some auxiliary definitions.

Physical memory access rights The predicate mmu_{ph} takes a state σ , the privilege level pl , a physical address pa and an access permission $req \in \{rd, wt, ex\}$ and holds if the access permission is granted for physical address pa .

$$mmu_{ph}(\sigma, pl, pa, req) \Leftrightarrow \exists va. mmu(\sigma, pl, va, req) = pa$$

The ARMv7 MMU mediates accesses to the virtual memory, enabling or forbidding operations to virtual addresses. Intuitively, a physical address pa can be read (written) if it exists at least a virtual addresses va that can be read (written) and that is mapped to pa according with the current page tables.

Write-derivability We say that a state σ' is *write-derivable* from a state σ in privilege level pl if their memories differ only for physical addresses that are writable in pl .

$$wd(\sigma, \sigma', pl) \Leftrightarrow \forall pa. \sigma.mem(pa) \neq \sigma'.mem(pa) \Rightarrow mmu_{ph}(\sigma, pl, pa, wt).$$

According with the MMU configuration in σ , only a subset of physical addresses are writable ($mmu_{ph}(\sigma, pl, pa, wt)$). Write-derivability identifies the set of states that can be produced by changing the memory content of an arbitrary number of such physical addresses with arbitrary values.

MMU-equivalence We say that two states are *MMU-equivalent* if for any virtual address va the MMU yields the same translation and the same access permissions.

$$\sigma \equiv_{mmu} \sigma' \Leftrightarrow \forall va, pl, req. mmu(\sigma, pl, va, req) = mmu(\sigma', pl, va, req)$$

Informally, two states are *MMU-equivalent* if their MMUs are configured exactly in the same way.

MMU-safety Finally, a state is *MMU-safe* if it has the same MMU behaviour as any state with the same coprocessor registers whose memory differs only for addresses that are writable in $PL0$.

$$mmu_s(\sigma) \Leftrightarrow \forall \sigma'. \sigma.coregs = \sigma'.coregs \wedge wd(\sigma, \sigma', PL0) \Rightarrow (\sigma \equiv_{mmu} \sigma')$$

A state is *MMU-safe* if there is no way to change the MMU configuration (i.e. the page tables) by writing into addresses that are writable in non-privileged mode. That is the MMU configuration prevents non-privileged SW to tamper the page tables.

An ARMv7 processor that obeys the access privileges computed by the MMU satisfies the following two properties:

Property 8 (ARM-integrity). *Assume $\sigma \in \Sigma$ with $mode(\sigma) = PL0$. If $\sigma \rightarrow_{PL0} \sigma'$ and $mmu_s(\sigma)$ then $wd(\sigma, \sigma', PL0)$ and $\sigma.coregs = \sigma'.coregs$, i.e., unprivileged steps from MMU-safe states can only lead into write-derivable states and do not affect the coprocessor registers.*

Note, that the MMU-safety prerequisite is not redundant here, because single instructions in ARM may result in a series of write operations, e.g., for “store pair” and unaligned store instructions. If the MMU configuration was not safe from manipulation in unprivileged mode, then such a series of writes could lead to an intermediate MMU configuration granting more write permissions than the initial one and the resulting state would not be write-derivable from σ .

Property 9 (ARM-confidentiality). *Let $\sigma_1, \sigma_2 \in \Sigma$ with $mode(\sigma_1) = mode(\sigma_2) = PL0$, and let A contains all physical addresses accessible in σ_1 , i.e., $A \supseteq \{pa \mid \exists req. mmu_{ph}(\sigma_1, PL0, pa, req)\}$. Suppose that $\sigma_1.uregs = \sigma_2.uregs$, $\sigma_1.coregs = \sigma_2.coregs$, $\sigma_1 \equiv_{mmu} \sigma_2$, and $\forall pa \in A. \sigma_1.mem(pa) = \sigma_2.mem(pa)$. If $\sigma_1 \rightarrow_{PL0} \sigma'_1$, there exists σ'_2 such that $\sigma_2 \rightarrow_{PL0} \sigma'_2$, $mmu_s(\sigma_1)$, and $mmu_s(\sigma_2)$ then*

$$\sigma'_1.uregs = \sigma'_2.uregs, \sigma'_1.coregs = \sigma'_2.coregs \text{ and } \forall pa \in A. \sigma'_1.mem(pa) = \sigma'_2.mem(pa).$$

Intuitively, Property 9 establishes that in MMU-safe configurations unprivileged transitions only can access information stored in the registers and in the part of memory that is readable in *PL0* according to access permissions. Within this paper we take Properties 8 and 9 for granted. In [127] the authors validated the HOL4 ARMv7 model against these properties assuming an identity-mapped address translation. Extending the result for an arbitrary but MMU-safe page table setup is currently nearing completion.

A.5 The Memory Virtualization API

The memory virtualization API is designed for the ARMv7-A architecture¹ and assumes neither hardware virtualization extensions nor TrustZone [4] support. To properly isolate the trusted components from the untrusted guest, which hosts a commodity OS, the memory virtualization subsystem needs to provide two main functionalities:

- Isolation of memory resources used by the trusted components.
- Virtualization of the memory subsystem to enable the untrusted OS to dynamically manage its own memory hierarchy, and to enforce access restrictions.

The physical memory region allocated to each type of client is statically defined. Inside its own region the guest OS is free to manage its own memory, and the virtualization API does not provide any additional guarantees for the security of the guest OS kernel against attacks from its user processes. However, using trusted services such as a run-time monitor it is possible to provide provable security guarantees to the guest OS, for instance to enforce the W \oplus X policy or to secure software updates, as explained in Section A.12.

A.5.1 Memory Management

The virtual memory layout is defined by a set of page tables that reside in physical memory. The configuration of these page tables is security-critical and must not be directly manipulated by untrusted parties. At the same time, the untrusted Linux kernel needs to manage its memory layout, which requires constant access to the page tables. Hence the hypervisor must provide a secure access mechanism, which we refer to as virtualizing the memory subsystem.

We use direct paging [26] to virtualize the memory subsystem. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Once the page tables are activated, the hypervisor must guarantee that further updates are possible only via the virtualization API to modify, allocate and free the page tables.

¹In practice, the presented design also supports the ARMv6 and ARMv5 architectures.

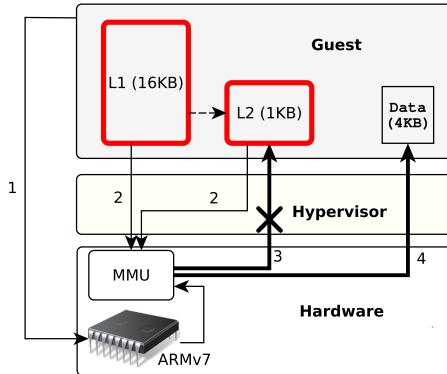


Figure A.2: Direct paging: 1) guest writes to virtual memory are mediated by the MMU as usual; 2) page tables are allocated in guest memory; 3) the hypervisor prevents writable mappings to guest memory regions holding page tables, forbidding the guest to directly modify them; 4) the hypervisor allows writable mappings to data blocks in guest memory.

Physical memory is fragmented into blocks of 4 KB. Thus, a 32-bit architecture has 2^{20} physical blocks. Since L1 and L2 page tables have size 16 KB and 1 KB respectively, an L1 page table is stored in four contiguous physical blocks and a physical block can contain four L2 page tables. We assign a type to each physical block, that can be:

- *data*: the block can be written by the guest.
- *L1*: contains part of an L1 and is not writable in unprivileged mode.
- *L2*: contains four L2 and is not writable in unprivileged mode.

The virtualization API shown in Figure A.3 is very similar to the MMU interface of the Secure Virtual Architecture [69] and consists of 9 hypercalls that select, create, free, map, or unmap memory blocks or page tables.

Figure A.2 indicates the address translation procedure and the connection between components of memory subsystem.

A.5.2 Enforcing The Page Type Constraints

Each API call needs to validate the page type, guaranteeing that page tables are write-protected. This is illustrated in Figure A.4. The table in the centre represents

<i>switch</i>	Select the active L1
<i>L1create</i>	Create page table of type L1
<i>L2create</i>	Create page table of type L2
<i>L1free</i>	Change the type of an L1 block to <i>data</i>
<i>L2free</i>	Change the type of an L2 block to <i>data</i>
<i>L1unmap</i>	Clear an entry of an L1 page table
<i>L2unmap</i>	Clear an entry of an L2 page table
<i>L1map</i>	Set an entry of an L1 page table
<i>L2map</i>	Set an entry of an L2 page table

Figure A.3: The virtualization API of the hypervisor to support direct paging.

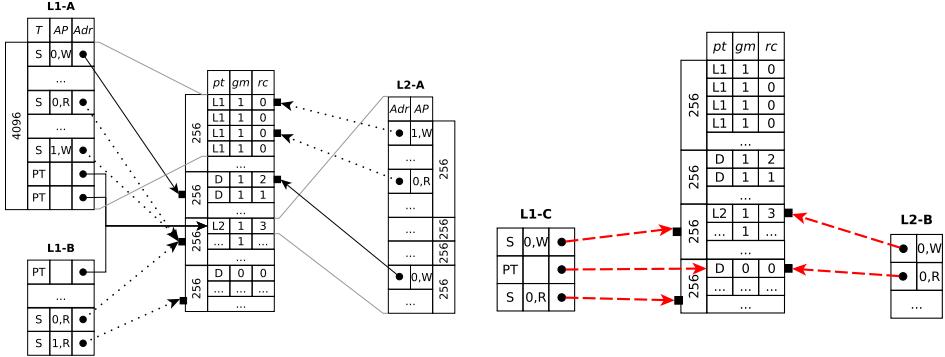


Figure A.4: Direct-paging mechanism. We use solid arrows to represent the L2 page table references and unprivileged write permissions, dotted arrows to represent other allowed references, and dashed arrows for references violating the page table policy.

the physical memory and stores the virtualization data structures for each physical block; the page type (*pt*), a flag informing if the block is allocated to guest memory (*gm*) and a reference counter (*rc*).

The four top most blocks contain an L1 page table, whose 4096 entries are depicted by the table *L1-A*. The top entry of the page table is a section descriptor ($T = S$) that grants write permission to the guest ($AP = (0, w)$). This entry's address component (*Adr*) points to the second physical section, which consists of 256 physical blocks. Two more section descriptors of L1-A are depicted in the table: the first one grants read-only permission to the guest ($(0, r)$), the second descriptor prevents any guest access and enables write permission for the privileged mode ($(1, w)$). The last two entries of L1-A are PT-descriptors. Each entry points to an L2 page table in the same physical block described by table L2-A and containing four L2 page tables.

The API calls manipulating an L1 enforce the following policy:

Any section descriptor that allows the guest to access the memory must point to a section for which every physical block resides in the guest memory space. Moreover, if a descriptor enables a guest to write then each block must be typed *data*. Finally, all PT-descriptors must point to physical blocks of type L2.

The Figure depicts two additional L1 page tables; *L1-B* and *L1-C*. The type of a physical block containing *L1-B* can be transformed to *L1* by invoking *L1create*. On the other hand, a block containing *L1-C* is rejected by *L1create* since the block contains three entries that violate the policy. In fact,

- the first descriptor grants guest write permission over a section which has at least one non data block, in this case L2,
- the second section descriptor allows the guest to access a section of the physical memory in which there exists a block that is outside the guest memory, and
- the third entry is a PT-descriptor, but points to a physical block that is not typed L2.

The first setting clearly breaks MMU-safety, since the guest is now able to write directly to a page table, circumventing the complete mediation of MMU configurations by the hypervisor. The second situation compromises confidentiality and possible integrity of the system if the guest has write access to the block outside its own memory. The third issue may again break MMU-safety if the referenced block is a writable data block. In case the referenced block contains (part of) another L1 page table this setting can lead to unpredictable MMU behaviour, since the L1 page table entries have a different binary format than the expected L2 entries.

The table *L2-A* depicts the content of a physical block typed L2 that contains four L2 page tables, each consisting of 256 entries. Each hypercall that manipulates an L2 enforces the following policy:

If any entry of the four L2 page tables grants access permission to the guest then the pointed block must be in the guest memory. If the entry also enables guest write access then the pointed block must be typed *data*.

For example a block containing *L2-B* is rejected by *L2create*, since the block contains at least two entries that violate the policy and thus threaten MMU-safety and integrity (in case of the first entry shown) as well as confidentiality (in case of the second one).

A naive run-time check of the page-type policy is not efficient, since it requires to re-validate the L1 page table whenever the *switch* hypercall is invoked. To

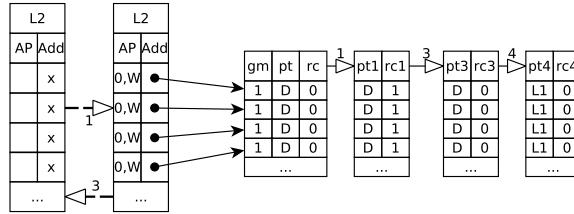


Figure A.5: Spawning a new process using the virtualization API. The guest (1) requests a writable mapping of four physical blocks to allocate a new L1 page table. After (2) setting up the table in memory (not shown), it asks (3) to remove the writable mapping, (4) to interpret the blocks as an L1 table, and (5) to make this one the new active L1 page table (not shown).

efficiently enforce that only *data* blocks can be written by the guest, the hypervisor maintains a reference counter, tracking for each block the sum of:

1. The number of descriptors providing writable access in user mode to the block.
2. The number of PT-descriptors that point to the block.

The intuition is that a hypercall can change the type of a physical block (e.g. allocate or free a page table) only if the corresponding reference counter is zero. Lemmas A.7.5 and A.7.6 in Section A.7 demonstrate that this approach is sound and that the page table policy described above is sufficient to guarantee MMU-safety.

In Figure A.5 we exemplify how an OS can use the API to spawn a new process. The OS selects four blocks from its physical memory to allocate a new L1 page table. We assume that initially the OS has no virtual mapping that enables it to access this part of the memory (i.e. the reference counter *rc* of these blocks is zero and the type *pt* is *data*).

1. Using *L2map*, the OS requests to change an existing L2 page table, establishing a writable mapping to the four blocks. The hypercall increases the reference counter accordingly (i.e. *rc1* = 1).
2. Without any mediation of the hypervisor, the OS uses the new mapping to write the content of the new L1 page table.
3. Using *L2unmap*, the guest removes the mapping established in (1) and decreases the reference counters (i.e. *rc3* = 0).
4. The guest invokes *L1create*, requesting the page table to be validated and the block type changed to L1. The request is granted only if the reference counter is zero, guaranteeing that there does not exist any mapping in the system that allows the guest to directly write the content of the page table.

5. Finally, the OS invokes *switch* to perform the context switch and to activate the new L1.

The example demonstrates some of the principles used to design the minimal API: (i) the address of the page tables are chosen by the guest, thus we do not need to change the OS allocators, (ii) the preparation of the page table can be done by the OS without mediation of the hypervisor, (iii) the content of the page table is not copied into the hypervisor memory, thus reducing memory accesses and memory overhead and not requiring dynamic allocation in the hypervisor, (iv) tracking the reference counter is used to guarantee the absence of page tables granting the guest write access to another page table, thus we can allow context switches among all created L1s without needing to re-validate their content.

A.5.3 Integrity of the Hypervisor Memory Map

When an exception is raised, the CPU redirects execution flow to a fixed location according to the exception vector. In ARMv7, subsequent instructions are executed in privileged mode but under the same virtual memory mapping as the interrupted guest. The hypervisor must enforce that the memory mapping of the exception vector, handler code, and hypervisor data structures is accessible during an exception without being modifiable by the guests. To this end, the hypervisor maintains its own static virtual memory mapping in a master page table and mirrors the corresponding regions to all L1s of the guest (with restricted access permissions).

A.5.4 Hypervisor Accesses to Guest Page Tables

The hypervisor APIs must be able to read and write the page tables allocated by the guest, in order to check the soundness of the requests and to apply the corresponding changes. The naive solution requires the hypervisor to change the current page table, enabling a hypervisor master page table whenever the guest memory must be accessed and then re-enabling the original page table before the guest is restored. This solution is expensive as it requires to flush TLB and caches. A solution tailored for Unixes can rely on the injective mapping built by the guest, which can be used by the hypervisor to access the guest kernel memory. In our settings the hosted guest is not trusted, thus this solution cannot guarantee that the injective mapping is obeyed by the guest. Some ARMv7 CPUs support special coprocessor instructions for virtual-to-physical address translation. These instructions can be used to validate the guest injective mapping at run-time. However, this approach is platform dependent and can result in nested exceptions that complicate the architecture and verification of the hypervisor. Instead, our design reserves a subset of the virtual address space for hypervisor use. The hypervisor master page table is built so that this address space is always mapped according to an injective translation (1-to-1) allowing the hypervisor to easily compute the virtual address for each physical address in the guest memory, similar to the direct memory maps

supported by FreeBSD [149] and Linux [73]. As with the hypervisor code and data structures, these regions are mirrored in all guest L1 tables.

A.5.5 Memory Model and Cache Effects

Hypervisors are complex software interacting directly with many low level hardware components, like processor, MMU, etc. Furthermore, there are hardware pieces that, while being invisible to the software layer, still can affect the system behaviour in many aspects. For example, the memory management unit relies on a caching mechanism, which is used to speed up accesses to page table entries. Basically, a data-cache is a shared resource between all partitions and it thus affects and is affected by activities of each partition. Consequently, data-caches may cause unintended interaction between software components running on the same processor, which can lead to cross-partition information leakage.

Moreover, for the ARMv7 architecture cache usage may cause sequential consistency to fail if the same physical address is accessed using different cacheability attributes. This opens up for TOCTTOU²-like vulnerabilities since a trusted agent may check and later evict a cached data item, which is subsequently substituted by an unchecked item placed in the main memory using an uncacheable alias. Furthermore, an untrusted agent can similarly use uncacheable address aliasing to easily measure which lines of the cache are evicted. This results in storage channels that are not visible in information flow analyses performed at the ISA level.

As an example (Figure C.4), the guest can use an uncacheable virtual alias of a page table entry in physical memory to bypass the page type constraints and install a potentially harmful page table. If the cache contains a valid page table entry PTE A for the physical address from a previous check by the hypervisor and this cache entry is clean (i.e., it will not be written back to memory upon eviction), the guest can (1) store an invalid (i.e. violating the page table policy) page table entry PTE B in a data page and (2) request the data page to become a page table. If the guest write is (3) directly applied to the memory, bypassing the cache using a uncacheable virtual address, and (4) the hypervisor accesses the same physical location through the cache, then the hypervisor potentially validates stale data (5). At a later point in time, the validated data PTE A is evicted from the cache and not written back to memory since it is clean. Then (6) the MMU will use the invalid page table containing PTE B instead and its settings become untrusted.

This kind of behaviour undermines the properties assured by formal verification that assumes a sequentially consistent model. In this paper, to counter this threat we use a naive solution; we prevent memory incoherence by cleaning the complete cache before accessing data stored by the guest. Clearly, this can introduce a substantial performance overhead, as shown in Section A.10.2. In [100], we demonstrate more efficient countermeasures to such threats and propose techniques to fix the verification.

²TOCTTOU – Time Of Check To Time Of Use

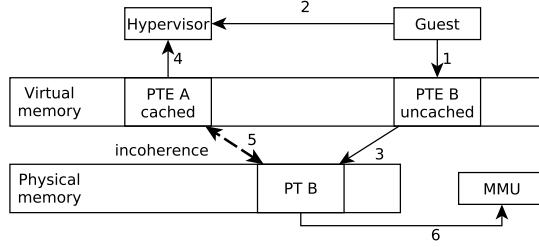


Figure A.6: Integrity threat due to incoherent memory caused by mismatched cacheability attributes. PTE A is validated by the hypervisor (4) but PTE B will be used as a page table entry for the guest (6).

A.6 Formalizing the Proof Goals

A.6.1 The Top Level Specification

A state of the Top Level Specification (TLS) is a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ and an abstract hypervisor state h . An abstract hypervisor state has the form $\langle pgtype, pgrefs \rangle$ where $pgtype$ indicates memory block types and $pgrefs$ maintains reference counters. Specifically, $pgtype \in 2^{20} \rightarrow \{D, L1, L2\}$ tracks the type of each 4 KB physical block; a block can either be (D) memory writable from the guest or data page, (L1) contain a L1 page table or (L2) contain a L2 page table. The map $pgrefs \in 2^{20} \rightarrow 2^{30}$ tracks the references to each physical block, as described in Section A.5.

The TLS interleaves standard unprivileged transitions with abstract handler invocations. Formally, the TLS transition relation $\langle \sigma, h \rangle \xrightarrow{i \in \{0,1\}} \langle \sigma', h' \rangle$ is defined as follows:

- If $\sigma \xrightarrow{PL0} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$; instructions executed in unprivileged mode that do not raise exceptions behave equivalently to the standard ARMv7 semantics and do not affect the abstract hypervisor state.
- If $\sigma \xrightarrow{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\langle \sigma, h \rangle \xrightarrow{1} H_a(\langle \sigma', h \rangle)$; whenever an exception is raised, the hypervisor is executed, modelled by the abstract handler H_a . The abstract handler always yields a state whose execution mode is unprivileged.

In our setup the trusted services and the untrusted guest are both executed in unprivileged mode. To distinguish between these two partitions, we use ARM domains. We reserve the domains 2-15 for the secure services.

Secure services Let $\sigma \in \Sigma$, the predicate $S(\sigma)$ identifies if the active partition is the one hosting the secure services: the predicate holds if at least one of the reserved domains (2-15) is enabled in the coprocessor registers $coregs$ of σ .

Intuitively, guaranteeing spatial isolation means confining the guest to manage a part of the physical memory available for the guest uses. In our setting, this part is determined statically and identified by the set of physical addresses G_m . Clearly, no security property can be guaranteed if the system starts from an insecure state; for example the guest must not be allowed to change the MMU behaviour by directly writing the page tables. For this reason we introduce a system invariant $I(\langle\sigma, h\rangle)$ that is used to constrain the set of secure initial states of the TLS. The set of all possible TLS states that satisfy the invariant is denoted by \mathcal{Q}_I . Then one needs to show:

Theorem A.6.1 (Invariant preserved). *Let $\langle\sigma, h\rangle \in \mathcal{Q}_I$ and $i \in \{0, 1\}$. If $\langle\sigma, h\rangle \rightarrow_i \langle\sigma', h'\rangle$ then $I(\langle\sigma', h'\rangle)$.*

Section A.7 elaborates the definition of the invariant and summarises the proof of the Theorem.

Complete mediation (MMU-integrity) is demonstrated by showing that neither the guest nor the secure services are able to directly change the content of the page tables and affect the address translation mechanism.

Theorem A.6.2 (MMU-integrity). *Let $\langle\sigma, h\rangle \in \mathcal{Q}_I$. If $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ then $\sigma \equiv_{mmu} \sigma'$.*

We use the approach of [108] to analyse the hypervisor data separation properties. The observations of the guest in a state $\langle\sigma, h\rangle$ is represented by the structure $O_g(\langle\sigma, h\rangle) = \langle\sigma.uregs, mem_g(\sigma), \sigma.coregs\rangle$ of user registers, guest memory $mem_g(\sigma)$ (the restriction of $\sigma.mem$ to G_m), and coprocessor registers. The latter are visible to the guest since they directly affect guest behaviour by controlling the address translation, and do not contain any information the guest should not be allowed to see. Evidently, however, all writes to the coprocessor registers must be mediated by the hypervisor.

The remaining part of the state (i.e. the content of the memory locations that are not part of the guest memory, banked registers) and, again, the coprocessor registers constitute the secure observations $O_s(\langle\sigma, h\rangle)$ of the state, which guest transitions are not supposed to affect.

The following theorem demonstrates that the context switch between the untrusted guest and the trusted services is not possible without the mediation of the hypervisor. The proof is straightforward, since S only depends on coprocessor registers that are not accessible in unprivileged mode.

Theorem A.6.3 (No context switch). *Let $\langle\sigma, h\rangle \in \mathcal{Q}_I$. If $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ then $S(\sigma) = S(\sigma')$.*

The *no-exfiltration* property guarantees that a transition executed by the guest does not modify the secure resources:

Theorem A.6.4 (No-Exfiltration). *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$.*

If $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ and $\neg S(\sigma)$ then $O_s(\langle \sigma, h \rangle) = O_s(\langle \sigma', h' \rangle)$.

The *no-infiltration* property is a non-interference property guaranteeing that guest instructions and hypercalls executed on behalf of the guest do not depend on any information stored in resources not accessible by the guest.

Theorem A.6.5 (No-Infiltration). *Let $\langle \sigma_1, h_1 \rangle, \langle \sigma_2, h_2 \rangle \in \mathcal{Q}_I$, $i \in \{0, 1\}$, and assume that $O_g(\langle \sigma_1, h_1 \rangle) = O_g(\langle \sigma_2, h_2 \rangle)$, $\neg S(\sigma_1)$, and $\neg S(\sigma_2)$.*

If $\langle \sigma_1, h_1 \rangle \rightarrow_i \langle \sigma'_1, h'_1 \rangle$ and $\langle \sigma_2, h_2 \rangle \rightarrow_i \langle \sigma'_2, h'_2 \rangle$ then $O_g(\langle \sigma'_1, h'_1 \rangle) = O_g(\langle \sigma'_2, h'_2 \rangle)$.

A.6.2 The Implementation Model

A critical problem of verifying low level platforms is that intermediate states of a handler that reconfigures the MMU can break the semantics of the high level language (e.g. C). For example, a handler can change a page table and (erroneously) unmap the region of virtual memory where the handler data structure (or the code) are located. For this reason we introduce the implementation model, that is sufficiently detailed to expose misbehaviour of the hypervisor accesses to the observable part of the memory (i.e. page tables, guest memory and internal data structures). The implementation interleaves standard unprivileged transitions and hypervisor functionalities. In contrast to the TLS, these functionalities now store their internal data in system memory, accessed by means of virtual addresses. In practice, in the implementation model the hypervisor functionalities are expressed as executable specifications, yet they are very close to the execution of the actual hardware at instruction semantics level. We demonstrate these differences by comparing two fragments of the TLS and the implementation specifications.

The TLS models the update of a guest page table entry as $\sigma'.mem = write_{32}(\sigma.mem, pa, desc)$, where pa is the physical address of the entry, $desc$ is a word representing the new descriptor and $write_{32}$ is a function that yields a new memory having four consecutive bytes updated. At the implementation level the same operation is represented as

```
if ⊥ mmu(σ, PL1, Gpa2va(pa), wt)
then ⊥
else write32(σ.mem,
    mmu(σ, PL1, Gpa2va(pa), wt), desc)
```

where mmu is the formal model of the ARMv7 MMU (introduced in Section A.4) and $Gpa2va$ is the function used by the hypervisor to compute the virtual address of a physical address that resides in guest memory. This function is statically defined and is the inverse of the injective translation established by the hypervisor master page table.

The implementation can fail to match the TLS for two reasons: (i) the current page table can prevent the hypervisor from accessing the computed virtual address, and then the implementation terminates in a failing state (denoted by \perp), (ii) the current address translation does not respect the expected injective mapping, thus $mmu(\sigma, PL1, Gpa2va(pa), wt) \neq pa$ and the implementation writes in an address that differs from the one updated by the TLS.

The next example shows the difference between accesses to the reference counter in the TLS and at implementation level. The TLS models this operation as $h.pgrefs(b)$, where b is the physical block. The implementation models the same operation using memory offsets as follows:

```
if  $\neg mmu(\sigma, PL1, tbl_{va} + 4*b, rd)$ 
then  $\perp$ 
else read32( $\sigma.mem$ ,
    mmu( $\sigma, PL1, tbl_{va} + 4*b, rd$ ))
    & 0xFFFFFFFF
```

This representation is directly reflected in the hypervisor code. For each block, the page type (two bits) and the reference counter (30 bits) are placed contiguously in a word. These words form an array, whose initial virtual address is tbl_{va} .

The handlers are represented by a HOL4 function H_r from ARMv7 states to ARMv7 states. The function is the executable specification of the various exception handlers including the MMU mapping/remapping/unmapping functionalities and is composed sequentially of the functional specifications for the corresponding code segments.

Then, the state transition relation $\rightarrow_{i \in \{0,1\}} \subseteq \Sigma \times (\Sigma \cup \{\perp\})$ determines the implementation behaviour as follows:

- If $\sigma \rightarrow_{PL0} \sigma'$ then $\sigma \rightarrow_0 \sigma'$; instructions executed in unprivileged mode that do not raise exceptions behave according to the standard ARMv7 semantics.
- If $\sigma \rightarrow_{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\sigma \rightarrow_1 H_r(\sigma')$; whenever an exception is raised, the hypervisor is executed and its behaviour is modelled by the function H_r . The function yields either a state whose execution mode is unprivileged or \perp .

To show implementation soundness we exhibit a refinement property relating abstract states $\langle \sigma_1, h \rangle$ to real states σ_2 . The refinement relation \mathcal{R} (that is left-total and surjective with the exception of the faulty state \perp) requires that: (i) the registers and coprocessors contain the same value in both states, (ii) the guest memory contains the same values in both states, (iii) the hypervisor data structures of the TLS state are projected into a part of hypervisor memory, and (iv) the reserved virtual addresses are always mapped in the same way as they are mapped in the master page table. Observations of the guest O_g^r and secure observations O_s^r are defined on real states using the hypervisor data structure mapping in analogy with

the corresponding observations on abstract states defined above. We require the refinement relation \mathcal{R} to be a bisimulation relation, that is preserved by computations of the abstract and implementation model.

Theorem A.6.6 (Implementation refinement). *Let $\langle \sigma_1, h \rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$ such that $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$. Let $i \in \{0, 1\}$.*

- *If $\sigma_2 \rightarrow_i \sigma'_2$ then exists $\langle \sigma'_1, h' \rangle$ such that $\langle \sigma_1, h \rangle \rightarrow_i \langle \sigma'_1, h' \rangle$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$.*
- *If $\langle \sigma_1, h \rangle \rightarrow_i \langle \sigma'_1, h' \rangle$ then exists σ'_2 such that $\sigma_2 \rightarrow_i \sigma'_2$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$.*

Finally we show that the security property of the TLS and the refinement relation directly transfer the mmu-integrity/no-exfiltration/no-infiltration to the implementation. We use Σ_I to represent the space of consistent implementation states: States σ_2 such that if $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ then $I(\langle \sigma_1, h \rangle)$.

Corollary 1 (Implementation security transfer). *Let $\sigma_1, \sigma_2 \in \Sigma_I$, $i \in \{0, 1\}$, $O_g^r(\sigma_1) = O_g^r(\sigma_2)$:*

- *if $\sigma_1 \rightarrow_0 \sigma'_1$ then $\sigma_1 \equiv_{mmu} \sigma'_1$*
- *if $\sigma_1 \rightarrow_0 \sigma'_1$ and $\neg S(\sigma_1)$ then $O_s^r(\sigma_1) = O_s^r(\sigma'_1)$*
- *if $\sigma_1 \rightarrow_i \sigma'_1$, $\sigma_2 \rightarrow_i \sigma'_2$, and $\neg S(\sigma_1)$ and $\neg S(\sigma_2)$ then $O_g^r(\sigma'_1) = O_g^r(\sigma'_2)$*

A.6.3 Binary Code Correctness

In the ARMv7 model of Section A.4 the untrusted guest, the trusted services and the hypervisor share the CPU, and the hypervisor behaviour is modelled by the execution of its binary instructions.

Intuitively, internal hypervisor states cannot be observed by the guest, since (i) during the execution of the handler the guest is not active, (ii) the hypervisor does not support preemption and (iii) the handlers do not raise nested exceptions. For this reason, we introduce a weak transition relation, which hides states that are privileged. We write $\sigma_0 \rightsquigarrow_i \sigma_n$ if there is a finite execution $\sigma_0 \rightarrow_i \dots \rightarrow \sigma_n$ such that $mode(\sigma_n) = PL0$ and $mode(\sigma_j) = PL1$ for $0 < j < n$.

Our goal is to exhibit a refinement property relating implementation states σ_1 and real states σ_2 . The refinement relation \mathcal{R}' (that is left-total with the exception of the faulty state \perp and surjective) requires that: (i) the registers and coprocessors contain the same value in both states, (ii) the guest memory contains the same values in both states, (iii) the memory holding the hypervisor data structures contains the same values in both states. For the observations of the guest O_g^r on real states, the same definition as for the implementation model are used, i.e. the guest can observe the same addresses in both models. Again the refinement is required to establish a bisimulation.

Theorem A.6.7 (Real Refinement). *Let $\sigma_1, \sigma_2 \in \Sigma_I$ such that $\sigma_1 \mathcal{R}' \sigma_2$. Let $i \in \{0, 1\}$.*

- If $\sigma_2 \rightsquigarrow_i \sigma'_2$ then exists σ'_1 such that $\sigma_1 \rightarrow_i \sigma'_1$ and $\sigma'_1 \mathcal{R}' \sigma'_2$.
- If $\sigma_1 \rightarrow_i \sigma'_1$ then exists σ'_2 such that $\sigma_2 \rightsquigarrow_i \sigma'_2$ and $\sigma'_1 \mathcal{R}' \sigma'_2$.

Finally one must show that the security properties are transferred to the real model.

Corollary 2 (Real Security Transfer). *Let $\sigma_1, \sigma_2 \in \Sigma_I$, $i \in \{0, 1\}$, $O_g^r(\sigma_1) = O_g^r(\sigma_2)$:*

- if $\sigma_1 \rightsquigarrow_0 \sigma'_1$ then $\sigma_1 \equiv_{mmu} \sigma'_1$
- if $\sigma_1 \rightsquigarrow_0 \sigma'_1$ and $\neg S(\sigma_1)$ then $O_s^r(\sigma_1) = O_s^r(\sigma'_1)$
- if $\sigma_1 \rightsquigarrow_i \sigma'_1$, $\sigma_2 \rightsquigarrow_i \sigma'_2$, and $\neg S(\sigma_1)$ and $\neg S(\sigma_2)$ then $O_g^r(\sigma'_1) = O_g^r(\sigma'_2)$

A.6.4 Execution Safety and End-to-End Information Flow Security

Note that we do not prove explicitly execution safety. The reason is that the transition relations of the ARM CPU and the TLS are left-total. Left-totality for the ARM CPU depends on the fact that the physical CPU never halts (with the exception of the privileged “wait” instruction that is never used by the hypervisor). Left-totality for the TLS holds because the virtualization API is modelled by HOL4 total functions over TLS states; every function is equipped with a termination proof, which is either automatically inferred by the theorem prover or has been manually verified. The only transitions that can yield a dead state (\perp) are the hypervisor transitions of the implementation model, due to incorrect memory accesses. Proving that this model can never reach the state \perp is part of the proof of Theorem A.6.6. It makes use of Lemma A.8.2, which shows that all hypervisor memory accesses are correct.

We do not prove standard end-to-end information flow properties because their definitions depend on the actual trusted services. This is often the case when two components are providing services to each other. For example, if the trusted service is the run-time monitor of Section A.12, then it should be able to directly read the memory of the untrusted Linux (to compute the signatures). Additionally, the trusted service can be allowed to affect the behaviour of the guest, for example by rebooting it or by changing its process table if a malware is detected.

However, our verification results enable the enforcement of end-to-end security by properly restricting the capability of the trusted services. In fact, these services are executed non-privileged, thus their execution is constrained by properties 8 and 9. Moreover, their memory mapping is static, is configured in the master page table of the hypervisor, and is independent of the guest configuration. If complete isolation is needed, it is sufficient to configure these entries of the master page table properly, use properties 8 and 9 together with Theorem A.6.2 to prove that the trusted services cannot affect and are independent of the guest resources. This

enables the trace properties to be established and consequently to obtain end-to-end security.

A.7 TLS Consistency

We proceed to describe the strategy for proving the TLS consistency properties of Section A.6.1. To this end we summarise the structure of the system invariant. The system invariant consists of two predicates: one (RC) ensures soundness of the reference counter, and the other (TC) guarantees that the state of the system is well typed.

The reference counter is sound (i.e. $RC(\sigma, h)$) if for every physical block b , the reference counter $h.pgrefs(b)$ is equal to $\sum_{i \in \{0 \dots 2^{20}-1\}} count(\langle\sigma, h\rangle, i, b)$, where $count$ is a function that counts the number of references from the block i to the block b , according to the reference-counter policy:

- if b is a data block and i is a page table, i.e. $h.pgtpe(b) = D$ and $h.pgtpe(i) \neq D$, then $count$ is the number of page table descriptors in i that are writable in non-privileged mode and that point to b ,
- if b is a L2 page table and i is a L1 page table then $count$ is the number of page table descriptors in i that use the table b to fragment the section, and
- if i is not a page table, i.e. $h.pgtpe(i) = D$, then $count(\langle\sigma, h\rangle, i, b) = 0$.

A system state is well typed ($TC(\sigma, h)$), if the MMU is enabled, the current L1 page table is inside a physical block of type L1, and each physical block b that does not have type *data* ($h.pgtpe(b) \neq D$) contains a sound page table ($sound(\langle\sigma, h\rangle, b)$) and resides in the guest memory ($pa \in G_m$ for all pa such that $pa[31 : 12] = b$). The predicate *sound* ensures that (i) no unpredictable setting is allowed, (ii) page tables grant write access only to blocks with type *data*, (iii) page tables forbid any access in $PL0$ to blocks outside the guest memory, and (iv) every L1 page table descriptor points to a block typed L2. Section A.5.2 and Figure A.4 exemplify these constraints.

The proofs of the theorems of Section A.6.1 have been obtained using the HOL4 theorem prover and the lemmas are described in the following.

Lemma A.7.1 (Invariant implies MMU-safety). *If $\langle\sigma, h\rangle \in \mathcal{Q}_I$ then $mmu_s(\sigma)$.*

Lemma A.7.1 demonstrates an important property of the system invariant; a state that satisfies the invariant has the same MMU behaviour as any state whose memory differs only for addresses that are writable in unprivileged mode. The proof of the lemma depends on the formal model of the ARMv7 MMU (but not on its instruction set); there the MMU behaviour is determined by coprocessor registers and the contents of the active L1 and referenced L2 page tables. The invariant guarantees that the active L1 page table of σ resides in four consecutive blocks that have type L1 and every page table descriptor in this table points to

a block typed L2. Moreover, only *data* blocks may be writable in unprivileged mode and write attempts to other blocks will be rejected. We examine a state σ' that is write-derivable in unprivileged mode from σ , but has the same coprocessor registers, selecting the same active L1 page table. Since the content of the page tables is unchanged, the MMU in σ' behaves exactly like in σ .

Proof sketch of Theorem A.6.2 To demonstrate MMU-integrity the ARM-integrity property is used. By definition of the TLS transition relation, if $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then (in the ARM model) $\sigma \rightarrow_{PL0} \sigma'$. Moreover, Lemma A.7.1 guarantees $mmu_s(\sigma)$. Thus, Property 8 can be used to conclude that $wd(\sigma, \sigma', PL0)$ and $\sigma.coregs = \sigma'.coregs$, i.e. σ' is a state write-derivable from σ and coprocessor registers have not changed. Finally, it suffices to apply Definition A.4 (MMU-safety) to show that $\sigma \equiv_{mmu} \sigma'$.

Lemma A.7.2 (Guest isolation). *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. For every physical address pa and access request req if $\neg S(\sigma)$ and $mmu_{ph}(\sigma, PL0, pa, req)$ then $G_m(pa)$.*

The proof of Lemma A.7.2 uses the formal ARMv7 MMU model and directly follows from the invariant. In particular, part (iii) of predicate *sound* demands that entries of a page table grant access permissions to the guest only if the entry points to a physical address that is inside the guest memory.

Proof sketch of Theorem A.6.4 Similar to the proof of Theorem A.6.2, the definition of the transition relation and Lemma A.7.1 yield that $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ implies $h = h'$, $\sigma \rightarrow_{PL0} \sigma'$ and $mmu_s(\sigma)$. Then Property 8 gives $\sigma.coregs = \sigma'.coregs$ and $wd(\sigma, \sigma', PL0)$, meaning that (according to the contraposition of Definition A.4) the memories of σ and σ' contain the same value for every physical address that is not writable in mode $PL0$ in σ . By Lemma A.7.2 the guest can only obtain write permissions to the physical addresses belonging to its own memory, thus the memories of σ and σ' have the same value for every physical address not in G_m . Moreover banked registers cannot be changed in unprivileged mode. Consequently, $O_s(\langle \sigma, h \rangle) = O_s(\langle \sigma', h' \rangle)$ holds as claimed.

Proof sketch of Theorem A.6.5 We proceed separately for unprivileged and privileged transitions. For unprivileged transition the ARM-confidentiality property is used. As proven above, from the definition of the transition relation, Lemma A.7.1, $\langle \sigma_1, h_1 \rangle \rightarrow_0 \langle \sigma'_1, h'_1 \rangle$, and $\langle \sigma_2, h_2 \rangle \rightarrow_0 \langle \sigma'_2, h'_2 \rangle$ we obtain $h_1 = h'_1$, $h_2 = h'_2$, $\sigma_1 \rightarrow_{PL0} \sigma'_1$, $\sigma_2 \rightarrow_{PL0} \sigma'_2$, $mmu_s(\sigma_1)$ and $mmu_s(\sigma_2)$. Since $O_g(\langle \sigma_1, h_1 \rangle) = O_g(\langle \sigma_2, h_2 \rangle)$, the user registers, guest memories (i.e. the content for addresses in G_m), and coprocessor registers are the same in σ_1 and σ_2 . The definition of $mmu_s(\sigma_1)$ yields $\sigma_1 \equiv_{mmu} \sigma_2$. Moreover, Lemma A.7.2 shows that the guest can obtain an access permission only to the physical addresses in G_m , thus the memories of σ_1 and σ_2 contain the same value for every address in $G_m \supseteq \{pa \mid \exists req. mmu_{ph}(\sigma_1, PL0, pa, req)\}$. This enables Property 9, which in turn justifies

that $\sigma'_1.uregs = \sigma'_2.uregs$ and $\forall pa \in G_m. \sigma'_1.mem(pa) = \sigma'_2.mem(pa)$, i.e. the guest observations in σ'_1 and σ'_2 are the same.

The proof of the Theorem A.6.5 for hypervisor transitions has been obtained by performing relational analysis. The function H_a accesses only three state components: the hypervisor data structures (i.e. h), the user registers and the memory (in order to validate page tables). The function H_a is symbolically evaluated on the states $\langle \sigma_1, h_1 \rangle$ and $\langle \sigma_2, h_2 \rangle$; whenever H_a updates an intermediate variable, it must be demonstrated that the value of the variable is the same in both executions, whenever H_a modifies a state component (e.g. memory or register), it must be demonstrated that the equivalence of guest observation is preserved. These tasks are completely automatic for assignments that only depend on intermediate variables and user registers. For every assignment that depends on memory accesses, a new verification condition is generated to require that the accessed addresses are the same in both executions and to guarantee that such address is in the guest memory. Finally, these verification conditions are verified, showing that H_a never accesses memory outside G_m .

Finally, we prove Theorem A.6.1 by showing that the invariant is preserved first by guest transitions (Lemma A.7.3) and then by the abstract handlers (Lemma A.7.4).

Lemma A.7.3 (Invariant vs guest). *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then $I(\langle \sigma', h' \rangle)$.*

This lemma demonstrates that the invariant is preserved by guest transitions. Its proof depends on the ARM-integrity property. It is straightforward to show that the invariant only depends on the content of the physical blocks that are not typed D and the hypervisor data (i.e. h and h'). Similar to the proof of Theorem A.6.4, the definition of the transition relation, Lemma A.7.1 and Property 8 guarantee that if $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ then $h = h'$, $\sigma \rightarrow_{PL0} \sigma'$, $mmu_s(\sigma)$ and $wd(\sigma, \sigma', PL0)$. As in the proof of Lemma A.7.1 it is shown that in σ every block that is not typed D is not writable, concluding that the invariant is preserved.

Lemma A.7.4 (Invariant vs hypervisor). *Let $\langle \sigma, h \rangle \in \mathcal{Q}_I$. If $\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle$ then $I(\langle \sigma', h' \rangle)$.*

The lemma demonstrates that the invariant is preserved by the handler functionalities and shows the functional correctness of the TLS design. By definition, if $\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle$ then there exists σ'' such that $\sigma \rightarrow \sigma''$, $mode(\sigma'') = PL1$ and $\langle \sigma', h' \rangle = H_a(\langle \sigma'', h \rangle)$. Similar to the proof of Lemma A.7.3, Property 8 is used to guarantee that the invariant is preserved by this transition: $I(\langle \sigma'', h \rangle)$. Then we show that the invariant is preserved by the abstract handler H_a .

This verification task requires the introduction of several supporting lemmas. The idea is that according to the input request, the abstract handler only changes a small part of the system state. For instance, when H_a maps a section, only the current L1 page table is modified, the contents of other blocks are unchanged. In

order to demonstrate that the invariant is indeed preserved for the parts of the state that are not affected by H_a , we introduce a number of additional lemmas. These lemmas are sufficiently general to be used to verify different virtualization mechanisms that involve direct paging and they prove the intuition that the type of a block can be safely changed when its reference counter is zero.

Definition Let h and h' be two abstract hypervisor states. The predicate $\text{type}_s(h, h')$ holds if and only if $h.\text{pgtype}(b) \neq h'.\text{pgtype}(b)$ implies $h.\text{refs}(b) = 0$ for all blocks b .

Changing the type of a block can affect the soundness of page tables that reference that block. The following lemma expresses the key property that soundness of page tables is preserved for all type changes of other blocks, as long as the reference counters of those blocks are zero:

Lemma A.7.5. *Assume $I\langle\sigma, h\rangle$ and $\text{type}_s(h, h')$. For every block b such that $h.\text{pgtype}(b) = h'.\text{pgtype}(b)$, if $\text{sound}(\langle\sigma, h\rangle, b)$ then $\text{sound}(\langle\sigma, h'\rangle, b)$.*

The proof of Lemma A.7.5 hinges on the fact that type changes can only break parts (ii) and (iv) of the page table soundness condition. However, if the type is only changed for blocks that are not referenced by any page table, soundness is preserved trivially.

We exemplify the usage of Lemma A.7.5 when proving Lemma A.7.4. Assume that H_a is allocating a new L2 page table in the block b' (i.e. changing the type of b' from D to $L2$). This operation can break soundness of any other block b . In fact, b can be a page table containing a writable mapping to b' , thus b is sound in $\langle\sigma, h\rangle$ but is unsound in $\langle\sigma, h'\rangle$. The side condition $\text{type}_s(h, h')$ ensures that this case cannot occur: to safely allocate a new page table, the reference counter of b' must be zero, thus b cannot contain a writable mapping to b' .

Similarly, the following lemma shows that, if the page type is changed only for blocks with zero references, then for all other page tables, the number of references is unchanged.

Lemma A.7.6. *Assume $I\langle\sigma, h\rangle$ and $\text{type}_s(h, h')$. For all blocks b, b' if $h.\text{pgtype}(b) = h'.\text{pgtype}(b)$ then $\text{count}(\langle\sigma, h\rangle, b, b') = \text{count}(\langle\sigma, h'\rangle, b, b')$.*

Finally we use the following lemma to show that the well-typedness of a block and its counted outgoing references are independent from the content of the other physical blocks.

Lemma A.7.7. *Let $\sigma, \sigma' \in \Sigma$ such that $I\langle\sigma, h\rangle$. If σ and σ' have the same memory content for the block b then $\text{sound}(\langle\sigma', h\rangle, b)$ and for every block b' $\text{count}(\langle\sigma', h\rangle, b, b') = \text{count}(\langle\sigma', h\rangle, b, b')$.*

For every functionality of the virtualization API (see Figure A.3), Lemmas A.7.5, A.7.6 and A.7.7 help to limit the proof of Lemma A.7.4 to only checking the well-typedness and soundness of the reference counter for the blocks that are affected by H_a .

Proof of Theorem A.6.1 The theorem directly follows from Lemmas A.7.3 and A.7.4.

A.8 Refinement

To verify the implementation refinement relation (i.e. prove Theorem A.6.6) we proved two auxiliary lemmas:

Lemma A.8.1 (Real MMU). *Let $\langle \sigma_1, h \rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$. If $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ then $\sigma_1 \equiv_{mmu} \sigma_2$.*

The Lemma shows that TLS and implementation states have the same MMU configuration. Its proof uses that the system invariant requires page tables to be allocated inside the guest memory, whose content is the same in the TLS and implementation states. Moreover, coprocessor registers contain the same data.

Lemma A.8.2 (Hypervisor page tables). *Let $\langle \sigma_1, h \rangle \in \mathcal{Q}_I$ and $\sigma_2 \in \Sigma$. If $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ then:*

1. *For all pa and req , if $pa \in G_m$ then $mmu(\sigma_2, Gpa2va(pa), PL1, req) = pa$.*
2. *For every block b and access request req , $mmu(\sigma_2, tbl_{va} + 4 * b, PL1, req) = tbl_{pa} + 4 * b$, where tbl_{pa} is the physical address where the hypervisor data structure is allocated.*

The Lemma shows that the implementation is always able to access the guest memory and the hypervisor data structures, and that the computed physical addresses match the expected values.

Proof sketch of Theorem A.6.6 To prove that the refinement is preserved by all possible transitions we verify independently the guest and hypervisor transitions. For guest transitions, Theorem A.6.4 (No-Exfiltration) and Lemma A.7.1 (MMU-safety) guarantee that the guest can change neither the memory outside G_m nor the page tables. Thus it is sufficient to show that the physical addresses of the hypervisor data structures are outside the guest memory. Moreover, Theorem A.6.5 (No-Infiltration) guarantees that the guest transition is not affected by a part of the state that is not equivalent in $\langle \sigma_1, h \rangle$ and σ_2 . For the hypervisor transition we used a compositional approach. First, we verified that all low-level operations (i.e. reads and updates of the page tables, reads and updates of the hypervisor data structures) preserve the refinement relation. Then we compose these results to show that the TLS and implementation transitions behave equivalently.

Proof sketch of Corollary 1 The proof depends on the fact the the relation \mathcal{R} is left-total and surjective. Proving that the security properties of the TLS are transferred to the implementation model is simplified by the definition of the refinement relation. For example, Lemma A.8.1 and Theorem A.6.2 are used to show that the MMU configuration cannot be changed by the untrusted guest. Assume $\sigma_2 \rightarrow_0 \sigma'_2$ and let $\langle \sigma_1, h \rangle$ be a TLS state such that $\langle \sigma_1, h \rangle \rightarrow_0 \langle \sigma'_1, h' \rangle$ and $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$. Since the refinement is preserved by all transitions (Theorem A.6.6), exists $\langle \sigma'_1, h' \rangle$ such that $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$. Lemma A.8.1 yields $\sigma_1 \equiv_{mmu} \sigma_2$ and Theorem A.6.2 (MMU-integrity) guarantees that $\sigma_1 \equiv_{mmu} \sigma'_1$, thus $\sigma_2 \equiv_{mmu} \sigma'_1$. Finally, Lemma A.8.1 yields $\sigma'_1 \equiv_{mmu} \sigma'_2$, thus $\sigma_2 \equiv_{mmu} \sigma'_2$. Similar reasoning is used to prove that properties *no-exfiltration* and *no-infiltration* are transferred to the implementation model, by showing that, if two TLS states have the same observations (i.e. $O_g(\langle \sigma_1, h \rangle) = O_g(\langle \sigma'_1, h' \rangle)$ or $O_s(\langle \sigma_1, h \rangle) = O_s(\langle \sigma'_1, h' \rangle)$) and the states are refined by two implementation states (i.e. $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ and $\langle \sigma'_1, h' \rangle \mathcal{R} \sigma'_2$), then the two implementation states have the same observations (i.e. $O_g^r(\sigma_2) = O_g^r(\sigma'_2)$ or $O_s^r(\sigma_2) = O_s^r(\sigma'_2)$).

A.9 Binary Verification

Binary analysis is key requirement to ensure security of low-level software platform, like hypervisors. Machine code verification obviates the necessity of trusting the compilers. Moreover, low level programs mix structured code (e.g. implemented in C) with assembly and use instructions (e.g. mode switches and coprocessor interactions) that are not part of the high level language, thus making difficult to use verification tools that target user level code.

For our hypervisor the main goal of the verification of the binary code is to prove Theorem A.6.7. This verification relies on Hoare logic and requires several steps. The first step (Section A.9.2) is transforming the relational reasoning into a set of contracts for the hypervisor handlers and guaranteeing that the refinement is established if all contracts are satisfied. Let C be the binary code of one of the handlers, the contract $\{P\}C\{Q\}$ states that if the precondition P holds in the starting state of C , then the postcondition Q is guaranteed by C .

Then, we adopt a standard semi-automatic strategy to verify the contracts. First, the weakest liberal precondition $WLP(C, Q)$ is computed on the starting state, then it is verified that the precondition P implies the weakest precondition.

The second verification step (computation of weakest preconditions) can be performed directly in HOL4 using the ARMv7 model. However, this task requires a significant engineering effort. We adopted a more practical approach, by using the Binary Analysis Platform (BAP) [47]. The BAP tool-set provides platform-independent utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to perform weakest-precondition calculations. These utilities reason on the BAP Intermediate Language (BIL), a

small and formally specified language that models instruction evaluation as compositions of variable reads and writes in a functional style.

The existing BAP front-end to translate ARM programs to BIL lacks several features required to handle our binary code: Support of ARMv7, management of processor status registers, banked registers for privileged modes and coprocessor registers. For this reason we developed a new front-end, which is presented in Section A.9.3, that converts an ARMv7 assembly fragment to a BIL program.

The final verification step consists of checking that the precondition P implies the weakest precondition. This task can be fully automated if the predicate $P \Rightarrow WLP(C, Q)$ is equivalent to a predicate of the form $\forall \vec{x}. A$ where A is quantifier free. The validity of A can then be checked using a Satisfiability Modulo Theory (SMT) solver that supports bitvectors to handle operations on words. In this work, we used STP [84].

An alternative approach for binary verification is to use the “decompilation” procedure developed by Myreen [155]. This procedure takes an ARMv7 binary and produces a HOL4 function that behaves equivalently (i.e. implements the same state transformation). This result allows to lift the verification of properties of assembly programs to reasoning on HOL4 functions. However, the latter task can be expensive due to the lack of automation in HOL4.

A.9.1 Soundness of the Verification Approach

The procedure described here to establish the functional correctness of the hypervisor code relies on four main arguments.

1. The HOL4 procedures that evaluate the effects of a given instruction in the ARMv7 model specify the updates to the processor state correctly. We use the ARMv7 step theorems to guarantee the correctness of this task.
2. The lifter transforms this state update information into an equivalent list of single-variable assignments in BIL. The correctness of this part of the lifter is an assumption for now.
3. The expressions in each update of a processor component are correctly translated to BIL expressions in the list of assignments, preserving their semantics. This has been proven for our lifter.
4. The binary code fragment that is lifted is actually executed on the ARMv7 hardware.

The last argument relies on the fact that the boot loader places the unmodified hypervisor image to the right place in memory. This is another assumption since we do not verify our boot loader. Furthermore there must not be self-modifying code. The easiest way to enforce this is to partition the hypervisor memory via its page table into data and code region and prove an invariant that the first is non-executable but the latter is non-writeable. There is no such protection against

self-modifying code in the hypervisor at the moment. Finally, one needs to show that the binary code is not interrupted, thus proving that the hypervisor is indeed non-preemptive. We do not have a full proof of the statement, but there are provisions in the lifter to show the absence of ARMv7 interrupts and exceptions.

For system call and unknown instructions, the lifter generates BIL statements that always fail, i.e., one can only verify programs in BAP that do not use such instructions. We follow the same approach for fetches, jumps, and memory instructions accessing constant addresses which are not mapped in the hypervisor’s static page table. Thus such operations cannot produce pre-fetch or data aborts. Additional care has to be taken to distinguish data and code regions to avoid permission faults due to writes to the code region or fetches from the data region, however there are no such checks at the moment. Indirect jumps are solved dynamically based on the lifted BIL program (see Sect. A.9.4) and for any jump to a location that is not defined in the program, i.e., not in the region accessible by the hypervisor, analysis with BAP will give an error. For dynamic memory accesses the lifter is able to insert assertions that the corresponding address is mapped, however the feature is currently not activated automatically. At last, the reception of external interrupts should not be re-enabled during hypervisor execution. Currently this invariant is not checked in the code verification but it could be easily added as an assertion between every instruction.

A.9.2 The Contracts Generation

Let C be the binary code of one of the handlers, we define the precondition P and the postcondition Q such that the contract subsumes the refinement:

- $P(\sigma_2) = \text{exists } \sigma_1 \text{ such that } \sigma_1 \mathcal{R}' \sigma_2$
- $Q(\sigma'_2, \sigma_2) = \text{for all } \sigma_1, \sigma'_1 \text{ if } \sigma_1 \mathcal{R}' \sigma_2 \text{ and } \sigma_1 \rightarrow_1 \sigma'_1 \text{ then } \sigma'_1 \mathcal{R}' \sigma'_2$

These contracts are not directly suitable for the verification of the binary code because the contracts quantify on states (σ_1 and σ'_1) that are in relation with the pre-state (σ_2) and post-state (σ'_2) of the binary code. We developed an HOL4 inference procedure specific for the structure of our hypervisor. The output of the procedure is a proof guaranteeing that the original contract $\{P\}C\{Q\}$ is satisfied if a “simplified” contract $\{P'\}C\{Q'\}$ is met. That is, for every σ_2, σ'_2 the predicate $P'(\sigma_2) \Rightarrow Q'(\sigma'_2, \sigma_2)$ implies $P(\sigma_2) \Rightarrow Q(\sigma'_2, \sigma_2)$.

This procedure makes heavy use of the simplification rules and decision procedures of HOL4. We informally summarise how this procedure works for the memory resource. The precondition P' is generated by transferring the hypervisor invariant I from the abstract model down to the real model. This is possible because (i) \mathcal{R}' constrains the memory holding the hypervisor data structures to be the same in σ_2 and σ_1 , (ii) \mathcal{R} (the refinement between the abstract model and the implementation model) guarantees that this memory area contains a projection of the hypervisor

data structures in the TLS state, (iii) on the TLS state the hypervisor invariant holds.

For the postcondition Q' we proceed as follows. If $\sigma_1 \rightarrow_1 \sigma'_1$ then $\sigma'_1 = H_r(\sigma_1)$. Let A be the set of memory addresses that are constrained by the refinement relation \mathcal{R}' and let B be the set of addresses that are modified by H_r . The set B is usually easy to identify in HOL4, thanks to its symbolic execution capability and the lemmas that have been already proven for the tasks of Section A.8. For each handler we demonstrate that $B \subseteq A$.

For every address $a \in (\bar{B}) \cap A$ (namely addresses constrained by the refinement relation that are not updated) we add to Q' the constraint $\sigma'_2.mem(a) = \sigma_2.mem(a)$. This uses $\sigma_1.mem(a) = \sigma_2.mem(a)$ and the refinement for the address a ($\sigma'_1.mem(a) = \sigma'_2.mem(a)$).

For every address $a \in B$ we make use of the HOL4 rewriting engine to obtain a naive symbolic execution of the handler specification. We use HOL4 to symbolically compute $H_r(\sigma_1)$ then we use the precondition $\sigma_1 \mathcal{R}' \sigma_2$ to rewrite the result and make sure that this is expressed only in terms of σ_2 . Let exp be the resulting expression, we add to Q' the constraint $\sigma'_2.mem(a) = exp.mem(a)$.

When the symbolic execution is too complex (e.g. too many outcomes are possible according to the initial state), we split the verification by generating multiple contracts $\{P_1\}C\{Q_1\}, \dots, \{P_n\}C\{Q_n\}$, where $P_i = P(\sigma_2) \wedge A_i(\sigma_2)$ and $\bigvee_i A_i$ is a valid formula (i.e. all possible cases are taken into account).

A.9.3 Translation of ARMv7 to BIL

The target language of the ARMv7 BAP front-end is BIL, a simple single-variable assignment language tailored to model the behaviour of assembly programs and developed to be platform independent. A BIL program is a sequence of statements. Each statement can affect the system state by assigning the evaluation of an expression to a variable, (conditionally or unconditionally) modifying the control flow, terminating the system in a failure state if an assertion does not hold and unconditionally halting the system in a successful state. The BIL data types for expressions and variables include boolean, words of several sizes and memories. The main constituent of BIL statements are expressions, that include constants, standard bit-vector binary and unary operators, and type casting function. Additionally, an expression can read several words from a memory or generate a new memory by changing a word in a given one.

We developed the new front-end on top of the HOL4 ARM model (see Section A.4), so that the soundness of the transformation from an ARM assembly instruction to its corresponding BIL program relies on the correctness of the ARM model used in HOL4 and not on a possibly different formalization of ARMv7. Our approach is illustrated in Figure A.7.

The HOL4 model provides a mechanism to statically compute the effects of an instruction via the *arm_steps* function. Let *inst* be the encoding of an instruction, then *arm_steps(inst)* returns the possible execution steps $\{st_1, \dots, st_n\}$. Each step

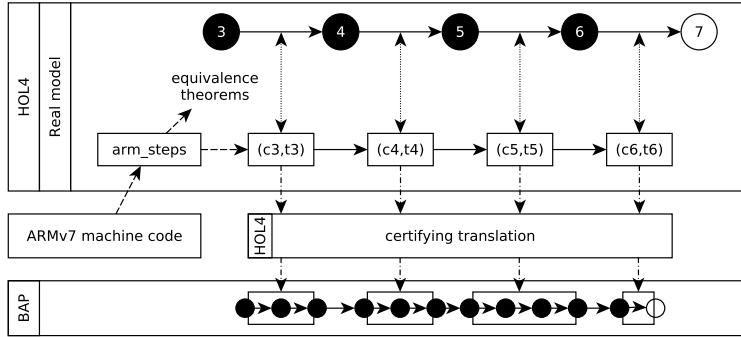


Figure A.7: Lifting machine code to BIL using the HOL4 ARMv7 model. The *arm_steps* function translates machine instructions into steps st_i consisting of guards c_i and transition functions t_i . Their effect is equivalent to the hypervisor computation in the real model (states 3-7, cf. Fig. A.1). Each step st_i is in turn translated into equivalent BIL code.

$st_i = (c_i, t_i)$ consists of the condition c_i that enables the transition and the function t_i that transforms the starting state into the next state. The function *arm_steps* is a certifying HOL4 procedure, since its output is a theorem demonstrating that for every $\sigma \in \Sigma$ if $fetch(\sigma) = inst$ and $c_i(\sigma)$ holds then $\sigma \rightarrow_{PL1} t_i(\sigma)$. For standard ARM decoding the function *fetch* reads four bytes from memory starting from the address pointed to by the program counter.

The translation procedure involves the following steps, (i) mapping HOL4 ARM states to BIL states and (ii) for each instruction of the given assembly fragment producing the BIL fragment that emulates the *arm_steps* outputs. To map an ARM state to the corresponding BIL state we use a straightforward approach. A BIL variable is used to represent a single component of the machine state: for example, the variable *R0* represents the register number zero and the variable *MEM* represents the system memory.

To transform an ARM instruction to the corresponding BIL fragment we need to capture all the possible effects of its execution in terms of affected registers, flags and memory locations. The generated BIL fragment should simulate the behaviour of the instruction executed on an ARM machine. Therefore, to obtain a BIL fragment for an instruction we need to translate the predicates c_i and their corresponding transformation functions t_i . This task is accomplished using symbolic evaluation of the predicates and the transformation functions. The input of the evaluation is a symbolic state in which independent variables are used to represent each state register, flag, coprocessor register and memory. This approach allows us to obtain a one-to-one mapping between the symbolic state variables and the BIL state variables. To transform a predicate c_i , we apply the predicate to a symbolic

ARMv7 state, thus obtaining a symbolic boolean expression in which free-variables are a subset of the symbolic state variables. Similarly, to map a transformation function t_i , we apply t_i to a symbolic state, thus obtaining a new state in which each register, flag and affected memory location is assigned a symbolic expression. Intuitively, for each instruction we produce the following BIL fragment:

```
label GUARD_1
cjmp | $b_1$ |, EFFECT_1, GUARD_2
...
label GUARD_N
cjmp | $b_n$ |, EFFECT_n, ERROR
label ERROR
assert false
```

Where ‘‘cjmp’’ is the BIL instruction for conditional jump and $|b_i|$ is a BIL boolean expression obtained by translating the symbolic evaluation of c_i . Then, for each step i we symbolically evaluate the transformation t_i and for each field (i.e. memory locations, registers, flags and coprocessor registers) that has been updated we transform the resulting symbolic expression and assign it to the corresponding BIL variable, generating a fragment

```
label EFFECT_i
var_1 = | $exp_1$ |
...
var_n = | $exp_n$ |
```

The described lifting procedure is straightforward. However, its soundness depends on the correct transformation of HOL4 terms (e.g. $|b_n|$ and $|exp_n|$) to BIL expressions. Since the number of HOL4 operators that occur in the generated expressions is huge, we cannot rely on a simple syntactical transformation to obtain a robust conversion of them to BIL. Moreover, the transformation of HOL4 terms to BIL expressions is used to convert the pre/post conditions of our contracts from HOL4 to BAP. For this reason, we formally modelled in HOL4 the BIL expression language (by providing a deep embedding of BIL expression in HOL4) and the translation procedure *liftExp* certifies its output:

$$\text{liftExp}(\textit{exp}) = (\textit{exp}', \vdash \textit{exp} = \textit{exp}')$$

In particular, the translation procedure yields a theorem demonstrating that the HOL4 input term \textit{exp} is equivalent to the BIL expression \textit{exp}' .

In order to dynamically generate the certifying theorem, the translation procedure is implemented in ML, which is the HOL4 meta language. The translation syntactically analyses and deconstructs the input expressions to select the theorems to use in the HOL4 conversion and rewrite rules. For terms composed by nested expressions the procedure acts recursively.

A.9.4 Supporting Tools

To compute the weakest precondition of a program is necessarily to statically know the control flow graph (CFG) of the program. This means that the algorithm depends on the absence of indirect jumps. Even if the hypervisor C-code avoids their explicit usage (e.g. by not using function pointers), the compiler introduces an indirect jump for each function exit point (e.g. the instruction at the address 0x20C in Figure A.8, is an indirect jump). Solving an indirect jump (i.e. enumerating all possible locations that can be target of the jump) is depending on checking the correctness of other properties of the application (e.g. the link register, which is usually used to track the return address of functions, can be pushed and popped from the stack, thus making the correctness of the control flow dependent on the integrity of the stack itself). Since we are interested in solving indirect jumps of code fragments that must respect contracts (Hoare triples $\{P\}C\{Q\}$), we implemented a simple iterative procedure that uses STP to discover all possible indirect jump targets under the contract precondition P .

1. The CFG of the of C fragment is computed using BAP. From the CFG, the list L of reachable addresses containing an indirect jump is extracted
2. For each address $a \in L$, the code fragment C is modified as follows:
 - a) let exp_a be the expression used in the indirect jump
 - b) the indirect jump is substituted with an assertion, which requires exp_a to be different from a fresh variable fv_a ; if such assertion fails, i.e. $exp_a = fv_a$, the modified fragment C terminates with a fault, otherwise it correctly terminates
3. the new fragment has no indirect jump; the weakest precondition WP of the postcondition $true$ (i.e. correct termination) is computed
4. the SMT solver searches for an assignment of the free variables (including all fv_i) that invalidates $P \Rightarrow WP$
5. if the SMT solver discovers a counterexample which involves the indirect jump at the address a , then it also discovers a possible target for this jump via selected assignment of the variable fv . Let exp be the expression used in the indirect jump. The fragment C is transformed by substituting the indirect jump with a conditional statement; if exp is equal to fv then jump to the fixed address fv , otherwise jump to the expression exp : `jmp exp` will be transferred into

```
c jmp exp == fv; value; new_label
label new_label: jmp exp
```

<pre> 0x100 bl #0x200 0x104 ... 0x108 bl #0x200 0x10C ... // function 0x200 push LR 0x204 str R1, R2 0x208 pop LR 0x20C b LR ... </pre>	<pre> label pc_0x100 PC = 0x100; LR = PC+4; jmp pc_0x100 label pc_0x104 ... label pc_0x108 PC = 0x108; LR = PC+4; jmp pc_0x100 label pc_0x10C ... label pc_0x200 PC = 0x200; mem=store32(mem, SP, LR); SP=SP-4 label pc_0x204 PC = 0x204; mem=store32(mem, R1, R2); label pc_0x208 PC = 0x208; LR=load32(mem, SP+4); SP=SP+4 label pc_0x20C PC = 0x20C; jmp LR </pre>
(a) Assembly	(b) BIL

Figure A.8: Indirect jump example

6. if the SMT solver does not find a counterexample, then every indirect jump is either unreachable or all its possible targets have been discovered. The fragment C is transformed by substituting every indirect jump with an assertion that always fails (`assert false`).
7. The procedure is restarted. Note that the inserted conditional statements prevent that the discovered assignments of fv_a can be used to invalidate the formula by the SMT solver in the next iteration.

In order to handle the greater complexity of the hypervisor code respect to the separation kernel verified in [71], we re-engineered this tool as a BAP plug-in. A particular problem that we face is that the CFG can contain loops if the same internal function of the hypervisor is called twice from different points in the program. Integrating the procedure with BAP allowed us to reuse the existing loop-unfolding algorithms to break these artificial loops.

We use figures A.8 and A.9 to demonstrate the algorithm. The assembly program (Figure A.8a) contains a function at `0x200`, which is invoked twice (from `0x100` and `0x108`). This function push the link register in the stack (`0x200`), writes the content of the register `R2` into the memory pointed by `R1` (`0x204`), pop the link register in the stack (`0x208`) and returns (`0x20C`). We assume that the precondition used is strong enough to ensure correct manipulation of the stack (e.g. the value of the stack pointer `SP` and the value of register `R1` used as pointer in the instruction

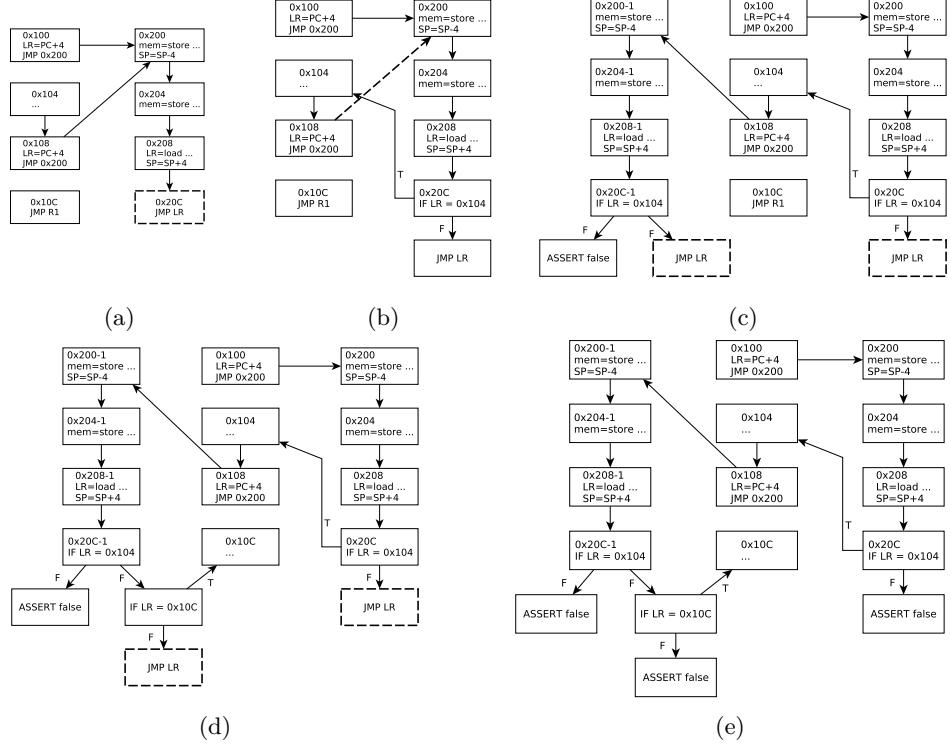


Figure A.9: Execution of the indirect jump solver

at `0x204` are distant at least one word). Figure A.8b and A.9a depict the BIL translation of the program and its initial CFG respectively. The CFG has only one reachable indirect jump (in `0x20C`), whose expression is `LR`. The SMT solver discovers a possible target for this jump (in this case `0x104`) and the program is transformed by substituting the indirect jump with a conditional statement, obtaining CFG is depicted in Figure A.9b. This CFG has an artificial loop due to the two invocations of the same function. Figure A.9c depicts the CFG obtained by unrolling the loop once. The program has now two reachable indirect jumps, the procedure is repeated and the SMT solver discovers that `0x10C` is a possible target of the jump in `0x20C-1`. The CFG is transformed as Figure A.9d. This CFG has still two indirect jumps. However, the SMT solver discovers that there is no assignment to the initial variables of the program that enables the activation of these jumps. Thus all indirect jumps have been resolved, the remaining ones are unreachable and are suppressed, obtaining the CFG in Figure A.9e.

In addition to solving indirect jumps, effective application of the verification strategy required the implementation of several tools and optimisation of the weak-

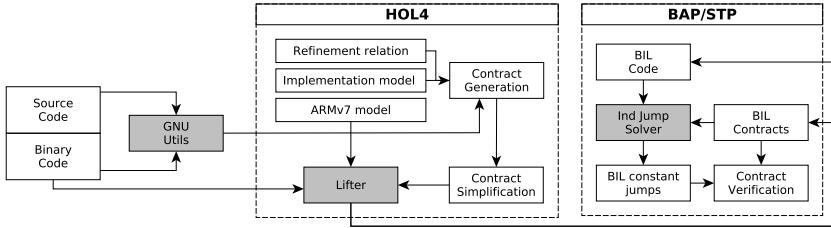


Figure A.10: Binary verification work-flow: *Contract Generation*, generating pre and post conditions based on the specification of the low-level abstraction and the refinement relation; *Contract Simplification*, massaging contracts to make them suitable for verification; *Lifter*, lifting handlers machine code and the generated contracts in HOL4 to BIL; *Ind Jump Solver*, procedure to resolve indirect jumps in the BIL code; *BIL constant jumps*, BIL fragments without indirect jumps; *Contract Verification* using SMT solver to verify contracts. Here, grey boxes are depicting the tools that have been developed to automate the verification as much as possible.

est precondition algorithm of BAP. Weakest preconditions can grow exponentially with regard to the number of instructions. Even though this problem cannot be solved in general, we can handle the most common case for ARM binaries, namely the sequential composition of several conditionally executed arithmetical instructions. This pattern matches the optimisation performed by the compiler to avoid small branches. We improved the BAP weakest precondition algorithm by adding a simplification function that identifies these cases. For some fragments of the code this straightforward strategy strongly reduced the size of the precondition; e.g. for one fragment consisting of 27 C lines compiled to 35 machine instructions the size of the precondition has been reduced from 8 GB to 15 MB.

Furthermore, machine code (and BIL) lacks information on data types (except for the native types like word and bytes) and represents the whole memory as a single array of bytes. Writing predicates and invariants is complex because their definition depends on location, alignment and size of data-structure fields. Moreover, the behaviour of compiled code often depends on the content of static memory used to represent constant values of the high level language. We developed a set of tools that integrate HOL4 and GDB to extract information from the C source code and the compiled assembly. With the support of these tools we are able to write the invariants and contracts of the hypervisor independently of the actual symbol locations and data structure offsets produced by the compiler.

Figure A.10 summarises the work-flow of our binary verification approach.

A.9.5 Limitations

The binary verification of the hypervisor has not been completed yet due to some time consuming tasks that require better automation. First, the inference procedure of Section A.9.2 uses the HOL4 simplification rules and decision procedures, however it is not completely automatic and must be adapted for every handler. Without taking into account the specificity of each handler, a naive procedure can easily generates contracts that cannot be handled by SMT solvers. For every handler, we manually specialize the procedure to generate contracts that have no quantifier in the precondition and only universal quantifiers in the postcondition.

Further complexity arises due to presence of loops. In theory, loops can be automatically handled by unfolding, since all loops in the hypervisor code iterate over fixed and limited ranges (e.g. the number of descriptors in a page table). Practically, this increases the size of the code (1024 times for handlers working on L2, and up to 4096*256 for handlers on L1) beyond the limit of programs that can be analyzed with BAP; thus the majority of loops must be manually handled.

By design, every loop in the hypervisor is also present in the specification. Let $C = C_1; \text{while}(B)\{C_2\}; C_3$ be the handler fragment and let $H_r(\sigma) = \text{let } \sigma_1 := H_1(\sigma) \text{ in let } \sigma_2 := \text{FOR}(b, H_2, \sigma_1) \text{ in } H_3(\sigma_2)$ be the specification. The problem of verifying that the refinement is preserved (i.e. if $\sigma \mathcal{R}' \sigma'$, and $C(\sigma)$ is the state produced by the program C , and $H_r(\sigma')$ is the state produced by the specification H_r then $C(\sigma) \mathcal{R}' H_r(\sigma')$) is reduced in verifying three refinements:

- $\sigma \mathcal{R}' \sigma'$ implies $C_1(\sigma) \mathcal{R}'_1 H_1(\sigma')$
- $\sigma \mathcal{R}'_1 \sigma'$ implies $C_2(\sigma) \mathcal{R}'_1 H_2(\sigma')$
- $\sigma \mathcal{R}'_1 \sigma'$ implies $C_3(\sigma) \mathcal{R}' H_3(\sigma')$

that is, a new refinement relation/invariant \mathcal{R}'_1 must be identified for the loop. This usually means identifying register allocation, allocations of variables on the stack etc. Due to lack of tools and integration with the compiler, this task is manually performed and requires to additionally specialize the inference procedure of Section A.9.2.

A.10 Implementation

The implementation of the hypervisor demonstrates the feasibility of our approach. The actual implementation targets BeagleBoard-xM (which is equipped with an ARM Cortex-A8) and supports the execution of Linux as the untrusted guest. The hypervisor executes both the untrusted guest and the trusted services in unprivileged mode, and their execution is cooperatively scheduled. Theorems A.6.1, A.6.2 and A.6.3 guarantee that the main security properties of the system (i.e. the correct setup of the page tables) cannot be violated by either the guest or the trusted services. Moreover, the untrusted guest cannot directly affect the trusted services

or directly extract information from their states (Theorems A.6.4 and A.6.5). This isolation is achieved by the complete mediation of the MMU settings and the allocation of the ARM domains 2-15 to the secure services. This approach limits the number of secure services to fourteen. However, this mechanism has the benefit of using the same page tables for both the guest and the trusted services (by reserving an area of the hypervisor virtual memory for the latter). This reduces the cost of context switch, since TLB and caches do not need to be cleaned. If more trusted services are needed, a separate page tables can be used.

The core of the hypervisor is the virtualization of the memory subsystem. This is provided by the handlers that are the subject of the verification and that are modelled by the transformations H_a and H_r (Section A.6). This core have been extended with additional handlers to provide further functionalities, which are needed to host a complete OS and to implement useful secure services. Since these additional handlers are not involved in the virtualization of the memory subsystem, establishing that they preserve the invariant (Theorem A.6.1) usually requires only to demonstrate that they do not directly change the physical blocks that contain the page tables and their memory safety.

A.10.1 Linux Support

The Linux kernel 2.6.34 has been modified to run on top of the hypervisor. This task required modification of architecture-dependent parts of the Linux kernel like execution modes, low-level exception routines and page table management. High-level OS functions such as process, resource and memory manager, file system, and networking did not require any modifications. This also introduce the additional handlers of the hypervisor that are not part of the formal verification.

CPU Privilege Modes In the absence of hardware supports, like virtualization extension, the target CPU includes only two execution modes: privileged and unprivileged (user). As for other approaches based on paravirtualization, since the hypervisor executes as privileged, the Linux kernel has been modified to execute as unprivileged. To separate kernel and user applications, the hypervisor manages two separate unprivileged execution contexts: virtual user and virtual kernel modes. In x86 these virtual modes can be implemented by segmentation. This approach is not possible for CPUs that do not provide this feature (e.g. x86 64-bit and ARM). Instead, we reserve the ARM domain 0 for the kernel virtual mode. Whenever the guest kernel requests a switch to virtual user mode (invoking the dedicated hypercall) we disable the domain 0, thus any access to the kernel virtual addresses generates a fault.

Note that the main security goal here is not to guarantee this OS-internal isolation, but to maintain the separation between the virtualized components (such as the Linux guest vs. secure data or services residing in non-guest memory).

CPU Exceptions CPU exceptions such as aborts and interrupts change the processor mode to privileged. These exceptions must therefore be handled in the hypervisor, which after validation can forward them to the unprivileged exception handlers of the Linux kernel. The hypervisor supplies the kernel exception handlers with some privileged data needed to correctly service an on-going exception (e.g. for pre-fetch abort, the privileged fault address and fault status registers are forwarded to the guest). The exception handlers in the Linux kernel have thus been slightly modified to support this. Among the exceptions that are forwarded to the Linux kernel there are the hardware interrupts delivered by the timer. This allows Linux to implement an internal time based scheduler.

Memory Management To paravirtualize the kernel, we modified the architecture dependent layer of its memory management. In the modified Linux all accesses to the coprocessor registers or to the active page tables are done by issuing the proper hypercalls. The architecture independent layer of the memory management has been left unmodified. In order to speed up the execution of Linux, a minimal emulation layer has been moved from the Linux kernel into the hypervisor itself. This layer reduces the overhead by translating a guest request into a sequence of invocations of the APIs that virtualize the MMU . Since the emulation layer accesses page tables only through the virtualization API, showing memory safety of this component is sufficient to extend the coverage of the verification.

A.10.2 Run-time Overhead

The port of the Linux kernel 2.6.34 on the hypervisor allows us to present a rough comparison of our approach with existing paravirtualized hypervisors for the ARM architecture [121]. The purpose of the evaluation is more to demonstrate that our approach actually runs with reasonable efficiency. A serious evaluation is out of scope of this work. It requires a more optimised implementation, and a more comprehensive evaluation.

The run-time evaluation is done using LMBench [151] running on Linux 2.6.34 with and without virtualization. The outcome, measured on an ARMv7-A Cortex-A8 system (BeagleBoard-xM [205]), is presented in Table A.2. The significant virtualization overhead for the fork benchmarks is due to a large number of simple operations (in this case, write access to a page-table) being replaced with a large number of expensive hypercalls. It may be possible to reduce this overhead with minimal optimisation (e.g. batching). In Table A.2 we also report measures from [121], where the authors compare several existing hypervisors for ARM. We point out that these performance numbers have been obtained from different sources, testing different ARM cores, boards and hosted Linux kernels. Hence we do not claim to be able to draw any hard conclusions from these figures about the relative performance of the hypervisors or their underlying architectures. With the purpose of demonstrating that the hypervisor can run efficiently real applica-

Benchmark	Hypervisor	Aggressive cache flushes	L4Linux	Xen	OKL4
null syscall	329%	332%	3043%	150%	60%
read	160%	181%	844%	90%	15%
write	193%	201%	877%	85%	24%
stat	83%	84%	553%	7%	
fstat	118%	122%	945%		42%
open/close	121%	119%	433%		
select(10)	78%	84%	4461%		14%
sig handler install	237%	245%	1241%		16%
sig handler overhead	226%	237%	1281%	82%	-14%
protection fault	40%	39%	975%		67%
pipe	168%	3073%	450%	74%	31%
fork+exit	195%	1861%	950%	247%	8%
fork+execve	187%	1787%	591%	239%	5%
pagefaults	435%	8740%	567%		

Table A.2: Latency benchmarks. LMBench micro benchmarks for the Linux kernel v2.6.34 running naively on BeagleBoard-xM, paravirtualized on the hypervisor without cache flushing (*Hypervisor*), with aggressive flushing (*Aggressive cache flushes*), and the other hypervisors (*L4Linux*, *Xen*, *OKL4*). Figures in the three last columns have been obtained from different ARM cores, boards and hosted Linux kernels

tions, we also measured the overhead introduced when executing tar, dd and several compression tools.

The second column reports the latency for the version of the hypervisor that aggressively flushes the caches (i.e. the caches are completely clean and invalidated whenever an exception or an interrupt is raised, while the hypervisor in the first column limits cache flushes to the cases of context switch). This naive approach guarantees that the actual CPU respects the fully sequential memory model, but introduces severe performance penalties especially in the application benchmarks. Less conservative approaches (e.g. evicting only the necessary physical addresses or forcing the page tables to be allocated in memory regions that are always cacheable) can be adopted for some processor implementations, but they require a more fine-grained modelling including caches and an adaptation of the verification approach for their justification, as discussed in [100].

A.10.3 Memory Footprint

The main difference between our proposal and the existing verified hypervisors is the MMU virtualization mechanism. The direct paging approach requires a table which contains at most $\text{mem_size}/\text{block_size}$ entries, where mem_size is the total available physical memory and block_size is the minimum page size (here, 4 KB). Each entry in this table uses $2 + \log_2 \max_{ref}$ bits, with the first two bits used to record entry type and \max_{ref} being the maximum number of references pointing to the same page. Assuming this number is bound by the number of processes, Table A.4 indicates the memory overhead introduced by direct paging.

It should be noted that on ARMv7, most operating systems including Linux dedicate one L1 page to each process and at least three L2 pages to map the stack,

Applications	Hypervisor	Aggressive cache flushes
tar (500 KB)	0%	171%
tar (1 MB)	0%	108%
dd (10 MB)	100%	1000%
dd (20 MB)	79%	932%
dd (40 MB)	76%	1061%
jpg2gif(5 KB)	0%	117%
jpg2bmp(5 KB)	0%	175%
jpg2bmp(250 KB)	0%	27%
jpg2bmp(750 KB)	-1%	24%
Jpegtrans(270', 5 KB)	0%	700%
Jpegtrans(270', 250 KB)	14%	300%
Jpegtrans(270', 750 KB)	11%	176%
Bmp2tiff(90 KB)	0%	500%
Bmp2tiff(800 KB)	0%	300%
Ppm2tiff(100 KB)	0%	600%
Ppm2tiff(250 KB)	0%	700%
Ppm2tiff(1.3 MB)	50%	350%
Tif2rgb(200 KB)	200%	1100%
Tif2rgb(800 KB)	25%	575%
Tif2rgb(1.200 MB)	31%	462%
sox(aif2wav -r 8000 -bits 16 100 KB)	50%	600%
sox(aif2wav -r 8000 -bits 16 500 KB)	75%	350%
sox(aif2wav -r 8000 -bits 16 800 KB)	83%	267%

Table A.3: Latency benchmarks. Application benchmarks for the Linux kernel v2.6.34 running natively on BeagleBoard-xM, paravirtualized on the hypervisor without cache flushing (*Hypervisor*), with aggressive flushing (*Aggressive cache flushes*).

Processes	Direct Paging		Shadow page table
	256 MB	1 GB	
32	56 KB	224 KB	608 KB
64	64 KB	256 KB	1216 KB
128	72 KB	288 KB	2432 KB

Table A.4: Memory footprint. Comparison of memory usage of *Shadow page table* and *direct paging*.

the executable code and the heap. Then the OS itself has a minimum footprint of $16\text{ KB} + 3 \times 1\text{ KB}$ per process. This footprint is doubled if the underlying hypervisor uses shadow page tables.

A.11 Evaluation

The hypervisor is implemented in C (and some assembly) and consists of 4529 lines of code (LOC). Excluding platform dependent parts, the hypervisor core is no larger than 2066 LOC. The virtualization of the memory subsystem consists of 1200 LOC. To paravirtualize Linux we changed 1025 LOC of its kernel, 950 in the ARM specific architecture folder and 75 in init/main.c. The paravirtualization is binary compatible with existing userland applications, thus we do not need to recompile either hosted applications or the libc. For comparison, the only other hypervisor that implements direct paging is the Xen hypervisor, which consists of 100 KLOC and its design is not suitable for verification. Instead, the small code base of our

hypervisor makes it easier to experiment with different virtualization paradigms and enables formal verification of its correctness. The formal specification consists of 1500 LOC of HOL4 and intentionally avoids any high level construct, in order to make the HOL4 model as similar as possible to the C implementation, at the price of increasing the verification cost. The complete proof consists of 18700 LOC of HOL4.

The verification highlighted a number of bugs in the initial design of the APIs: (i) arithmetic overflow when updating the reference counter, caused by not preventing the guest to create an unbounded number of references to a physical block, (ii) bit field and offset mismatch, (iii) missing check that a newly allocated page table prevents the guest to overwrite the page table itself, (iv) usage of the signed shift operator where the unsigned one was necessary and (v) approval of guest requests that cause unpredictable MMU behaviour. Moreover, the verification of the implementation model identified three additional bugs exploitable by the guest by requesting the validation of page tables outside the guest memory. Finally, the methodology described in Section A.9 has been experimented in the verification of the binary code of one of the hypercalls. This experiment identified a buffer overflow in the binary code that was missing in implementation model. The HOL4 model uses a 10-bit variable to store an untrusted parameter which is later used to index the entries of a page table. The binary code uses a 32-bit registers to store the same parameter, thus causing an overflow when accessing the L2 page table if the received parameter is bigger than 1023. The bug has been fixed by sanitising the input using the mask `parameter = parameter & 0x3ff`.

The project was conducted in three steps. The design, modelling and verification of the APIs for MMU virtualization required nine person months. Here, the most expensive tasks have been the verification of Theorems A.6.1 and A.6.6. The C implementation of the APIs and the Linux port has been accomplished in three person months. While the implementation team was completing the Linux port the verification team started the verification of the refinement, which has taken three months so far. This work is continuing, in order to complete the verification from the HOL4 implementation level down to assembly.

A.12 Applications

Applications of the hypervisors include the deployment of trusted cryptographic services and trusted controllers. In the first scenario, the hypervisor core is extended with the handlers required to implement message passing. These handlers allow (i) Linux to send a message to the trusted service, (ii) the trusted service to reply with an encrypted message and (iii) the two partitions to cooperatively schedule themselves. The isolation properties guarantee that the untrusted guest cannot access the cryptographic keys stored in the memory of the trusted services. The second scenario includes a device (e.g. a physical sensor) whose IO is memory mapped. The guest is forbidden to access the memory where the IO registers are

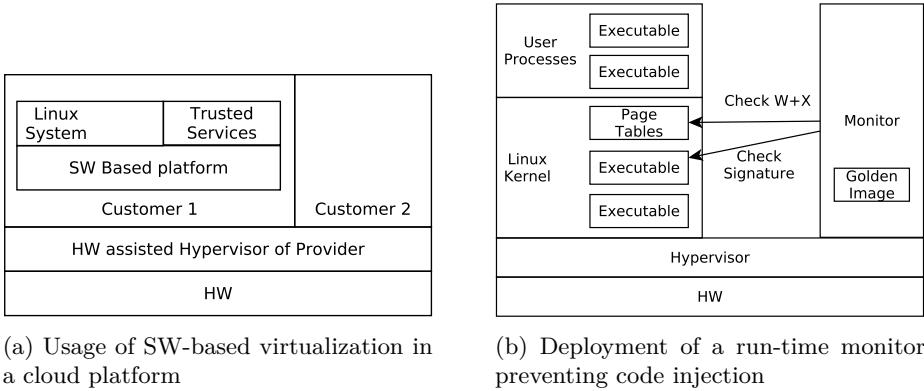


Figure A.11: Applications of the secure virtualization platform.

mapped, thus guaranteeing that the trusted controller is the only subject capable of directly affecting the device. The complete Linux system can be used to provide a rich and complex user interface (either graphical or web based) for the controller logic without affecting its security.

The MMU virtualization solution demonstrated here can be used by other ARM-based software platforms than the hypervisor reported above. A fully fledged hypervisor (e.g. XEN) can use our approach to support hardware that lacks virtualization extensions (e.g. Cortex-A8, Cortex-A5, ARM11). The mechanism can also be used by compiler-based virtual machines and unikernels, which need to monitor the memory configuration and protect it from the rest of the system (e.g. SVA uses a non-verified implementation of direct paging). Customers of cloud infrastructures can also benefit from our approach (see Figure A.11a). In this setting, if the virtualization extensions are available, the most privileged execution mode is controlled by the software platform managed by the cloud provider (e.g. a hypervisor). Thus, these extensions cannot be used by the customer to isolate its untrusted Linux from its own trusted services. In this setup, our mechanism can be used to fulfil this requirement.

An interesting application of isolating platforms is the external protection of an untrusted commodity OS from internal threats, as demonstrated in [66]. Trustworthy components are deployed together and properly isolated from the application OS (see Figure A.11b). These components are used as an aid for the application OS to restrict its own attack surface, by guaranteeing the impossibility of certain malicious behaviours. In [51], we show that this approach can be used to implement an embedded device that hosts a Linux system provably free of binary code injection. Our goal is to formally prove that the target system prevents all forms of binary code injection even if the adversary has full control of the hosted Linux and no analysis of Linux is performed.

The system is implemented as a run-time monitor. The monitor forces an

untrusted Linux system to obey the executable space protection policy (usually represented as $W \oplus X$); a memory area can be either executable or writable, but cannot be both. The protection of executable space allows the monitor to intercept all changes to the executable code performed by a user application or by the Linux kernel itself. On top of this infrastructure, we use standard signature checking to prevent code injection. Here, integrity of an executable physical block stands for the block having a valid signature. Similarly, the integrity of the system code depends on the integrity of all executable physical blocks. The valid signatures are assumed to be known by the run-time monitor. We refer to this information as the “golden image” (GI) and it is held by the monitor.

We configured the hypervisor to support the following interaction protocol:

1. For each hypercall invoked by a guest, the hypervisor forwards the request to the monitor.
2. The monitor validates the request based on its validation mechanism.
3. The monitor reports to the hypervisor the result of the hypercall validation.

Since the hypervisor supervises the changes of the page tables, the monitor is able to intercept all the memory layout modifications. This makes the monitor able to know whether a physical block is writable: if there exists at least one virtual mapping pointing to such block and having writable access permission. Similarly it is possible to know which physical block is executable.

Then the signature checking is implemented in the obvious way: whenever Linux requests to change a page table (i.e. causing to change the domain of the executable code) the monitor (i) identifies the physical blocks that can be made executable by the request, (ii) computes the block signature and (iii) compares the result with the content of the golden image. This policy is sufficient to prevent code injection that is caused by changes of the memory layout setting, due to the hypervisor forwarding to the monitor all requests to change the page tables.

Figure A.11b depicts the architecture of the system; both the run-time monitor and the untrusted Linux are deployed as two guests of the hypervisor. Using a dedicated guest on top of the hypervisor permits to decouple the enforcement of the security policies from the other hypervisor functionalities, thus keeping the trusted computing base minimal. Moreover, having the security policy wrapped inside a guest supports both the tamper-resistance and the trustworthiness of the monitor. In fact, the monitor can take advantage from the isolation properties provided by the hypervisor. This avoids malicious interferences coming from the other guests (for example from a process of an OS running on a different partition of the same machine). Finally, decoupling the run-time security policy from the other functionalities of the hypervisor makes the formal specification and verification of the monitor more affordable.

The formal model of the system (i.e. consisting of the hypervisor, the monitor and the untrusted Linux) is built on top of the models presented in Section A.6.1.

Here we leave unspecified the algorithm used to sign and check signatures, so that our results can be used for different intrusion detection mechanisms. The golden image GI is a finite set of signatures $\{s_1, \dots, s_n\}$, where the signatures are selected from a domain S . We assume the existence of a function $sig : 2^{4096*8} \rightarrow S$ that computes the signature of the content of a block. The system behaviour is modelled by the following rules:

1.
$$\frac{\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle}{\langle \sigma, h, GI \rangle \rightarrow_0 \langle \sigma', h', GI \rangle}$$
2.
$$\frac{\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle \quad validate(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \rightarrow_1 \langle \sigma', h', GI \rangle}$$
3.
$$\frac{\langle \sigma, h \rangle \rightarrow_1 \langle \sigma', h' \rangle \quad \neg validate(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \rightarrow_1 \epsilon(\langle \sigma, h, GI \rangle)}$$

User mode transitions (e.g. Linux activities) require neither the hypervisor nor the monitor intermediation. Theorem A.6.4 justifies the fact that, by construction, the transitions executed by the untrusted component cannot affect the monitor state; (i) the golden image is constant and (ii) the monitor code can be statically identified and abstractly modelled. The executions in privileged mode require the intermediation of the monitor. If the monitor validates the request, then the standard behaviour of the hypervisor is executed. Otherwise the hypervisor performs a special operation to reject the request, by reaching the state that is returned by a function ϵ . Hereafter, the function ϵ is assumed to be the identity. Alternatively, ϵ can transform the state so that the requestor is informed about the rejected operation, by updating the user registers according to the desired calling convention. The function $validate(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)$ represents the validation mechanism of the monitor, which checks at run-time possible violations of the security policies.

To formalize the top level goal of our verification we introduce some auxiliary notations. The “working set” identifies the physical blocks that host executable binaries and their corresponding content. Let σ be a machine state. The working set of σ is defined as

$$WS(\sigma) = \{\langle bl, content(bl, \sigma) \rangle \mid \exists pa. mmu_{ph}(\sigma, PL0, pa, ex) \wedge pa \in bl\}$$

By using a code signing approach, we say that the integrity of a physical block is satisfied if the signature of the block’s content belongs to the golden image. Let $cnt \in 2^{4096*8}$ be the 4 KB content of a physical block bl and GI be the golden image

$$int(GI, bl, cnt) = sig(bl, cnt) \in GI$$

Notice that our security property can be refined to fit different anti-intrusion mechanisms. For example, $int(GI, bl, cnt)$ can be instantiated with the execution of an anti-virus scanner.

The system state is free of malicious code injection if the signature check is satisfied for the whole executable code. That is: Let σ be a machine state, bl be a

physical block and GI be the golden image

$$\text{int}(GI, \sigma) \Leftrightarrow \forall \langle bl, cnt \rangle \in WS(\sigma) . \text{int}(GI, bl, cnt)$$

Finally, in [51] we demonstrate our top level goal: No code injection can succeed.

Theorem A.12.1. *If $\langle \sigma, h, GI \rangle$ is a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI \rangle$ then $\text{int}(GI, \sigma)$*

We implemented a prototype of the system. The monitor code consists of 720 lines of C and 100 lines have been added to the hypervisor to support the needed interactions among the hosted components.

A.13 Concluding Remarks

We have presented a memory virtualization platform for ARM based on direct paging, an approach inspired by the paravirtualization mechanism of Xen [26], and the Secure Virtual Architecture [69]. The platform has been verified down to a detailed model of a commodity CPU architecture (ARMv7-A), and we have shown a hypervisor based on the platform capable of hosting a Linux system while provably isolating it from other services. The hypervisor has been implemented on real hardware and shown to provide promising performance, although the benchmarks presented here are admittedly preliminary. The verification is done with respect to a top-level model that augments a real machine state with additional model components. The verification shows complete mediation, memory isolation, and information flow correctness with respect to the top-level model. As the main application we demonstrated how the virtualization mechanism can be used to support a provably secure run-time monitor for Linux that provides secure updates along with the W \oplus X policy.

The main precursor work on formally verified MMU virtualization uses the simulation-based approach of Paul et al [11, 164, 9]. In [11, 164] shadow page tables are used to provide full virtualization, including virtual memory, for “baby VAMP”, a simplified MIPS, using VCC. Full virtualization is generally more complex than the paravirtualization approach studied in the present paper, but the machine model is simplified, information flow security is not supported by the simulation framework, and neither applications nor implementation on real hardware are reported. In [9] the same simulation-based approach is used to study TLB virtualization on an abstract version of the x64 virtual memory architecture. Other related work on verification of microkernels and hypervisors such as seL4 [133] or the Nova project [199] does not address MMU virtualization in detail. It may be argued that the emergence of hardware based virtualization support makes software MMU virtualization obsolete. We argue that this is not the case. First, many platforms remain or are currently in development that do not yet support virtualization extensions, second, many application hardening frameworks such as Criswell et al. [68], KCoFi [66], Overshadow [49], Inktag [112] and Virtual Ghost [67] rely

on some form of MMU virtualization for their internal security, and third, some use cases, e.g. in cloud scenarios, could make good use of software based MMU virtualization to harden VMs without relying on cloud provider hardware.

Our results are not yet complete. The MMU virtualization approach does not support DMA. To securely enable DMA the behaviour of the specific DMA controller must be formally modelled (in [186] the authors describe a framework for such extensions and establish Properties 8 and 9 for the resulting model) and the hypervisor must (i) mediate all accesses to the memory area where the controller's registers are mapped, (ii) enable a DMA channel only if the pointed physical blocks is *data* and (iii) update the reference counters accordingly. Several embedded platforms are equipped with IOMMUs, that provide HW support to isolate/confine external peripherals that use DMA. However SW based isolation of DMA is still interesting since it can be used in the scenarios where these HW extensions are not available (e.g. CortexM microcontrollers), they are not accessible (e.g. when they are managed by a cloud provider), or in time critical applications since the page walks introduced by the IOMMU can slow down the peripheral and make worst case execution time analysis more difficult.

A tricky problem concern the treatment of unpredictable behaviour in the ARMv7 architecture. The Cambridge ISA model [82] maps transitions resulting in unpredictable behaviour to \perp . We ignore this for the following reason. Our verification shows that unpredictable behaviour never arises during hypervisor code execution. This is so since the ARMv7 step theorems used by the lifter are defined only for predictable instructions, and since our invariant guarantees that the MMU configuration is always well defined. As a result unpredictable behaviour can arise only during non-privileged execution, the analysis of which we have in effect deferred to other work [186].

Finally more work is needed to properly reflect caches, TLBs, and, further down the line, multi-core. The soundness of the current implementation depends on the type of data cache, and on flushing the cache when needed, in order to support a linearizable memory model. To enable more aggressive optimisation, and to fully formally secure our virtualization framework on processors with weaker cache guarantees, the model must be extended to reflect cache behaviour.

Paper B

Trustworthy Prevention of Code Injection in Linux on Embedded Devices

Hind Chfouka, Hamed Nemati, Roberto Guanciale, Mads Dam, Patrik Ekdahl

Abstract

We present MProsper, a trustworthy system to prevent code injection in Linux on embedded devices. MProsper is a formally verified run-time monitor, which forces an untrusted Linux to obey the executable space protection policy; a memory area can be either executable or writable, but cannot be both. The executable space protection allows the MProsper’s monitor to intercept every change to the executable code performed by a user application or by the Linux kernel. On top of this infrastructure, we use standard code signing to prevent code injection. MProsper is deployed on top of the Prosper hypervisor and is implemented as an isolated guest. Thus MProsper inherits the security property verified for the hypervisor: (i) Its code and data cannot be tampered by the untrusted Linux guest and (ii) all changes to the memory layout is intercepted, thus enabling MProsper to completely mediate every operation that can violate the desired security property. The verification of the monitor has been performed using the HOL4 theorem prover and by extending the existing formal model of the hypervisor with the formal specification of the high level model of the monitor.

B.1 Introduction

Even if security is a critical issue of IT systems, commodity OSs are not designed with security in mind. Short time to market, support of legacy features, and adoption of binary blobs are only few of the reasons that inhibit the development of secure commodity OSs. Moreover, given the size and complexity of modern OSs,

the vision of comprehensive and formal verification of them is as distant as ever. At the same time the necessity of adopting commodity OSs can not be avoided; modern IT systems require complex network stacks, application frameworks etc.

The development of verified low-level execution platforms for system partitioning (hypervisors [138, 157], separation kernels [174, 71], or microkernels [133]) has enabled an efficient strategy to develop systems with provable security properties without having to verifying the entire software. The idea is to partition the system into small and trustworthy components with limited functionality running alongside large commodity software components that provide little or no assurance. For such large commodity software it is not realistic to restrict the adversary model. For this reason, the goal is to show, preferably using formal verification, that the architecture satisfies the desired security properties, even if the commodity software is completely compromised.

An interesting usage of this methodology is when the trustworthy components are used as an aid for the application OS to restrict its own attack surface, by proving the impossibility of certain malicious behaviors. In this paper, we show that this approach can be used to implement an embedded device that hosts a Linux system provably free of binary code injection. Our goal is to formally prove that the target system prevents all forms of binary code injection even if the adversary has full control of the hosted Linux and no analysis of Linux itself is performed. This is necessary to make the verification feasible, since Linux consists of million of lines of code and even a high level model of its architecture is subject to frequent changes.

Technically, we use Virtual Machine Introspection (VMI). VMI is a virtualized architecture, where an untrusted guest is monitored by an external observer. VMI has been proposed as a solution to the shortcomings of network-based and host-based intrusion detection systems. Differently from network-based threat detection, VMI monitors the internal state of the guest. Thus, the VMI does not depend on information obtained from monitoring network packets which may not be accurate or sufficient. Moreover, differently from host-based threat detection, VMIs place the monitoring component outside of the guest, thus making the monitoring itself tamper proof. A further benefit of VMI monitors is that they can rely on trusted information received directly from the underlying hardware, which is, as we show, out of the attackers reach.

Our system, MProsper, is implemented as a run-time monitor. The monitor forces an untrusted Linux system to obey the executable space protection policy (usually represented as $W \oplus X$); a memory area can be either executable or writable, but cannot be both. The protection of executable space allows MProsper to intercept all changes to the executable code performed by a user application or by the Linux kernel itself. On top of this infrastructure, we use standard code signing to prevent code injection.

Two distinguishing features of MProsper are its execution on top of a formally verified hypervisor (thus guaranteeing integrity) and the verification of its high level model (thus demonstrating that the security objective is attained). To the best of

our knowledge this is the first time the absence of binary code injection has been verified for a commodity OS. The verification of the monitor has been performed using the HOL4 theorem prover and by extending the existing formal model of the hypervisor [157] with the formal specification of the monitor’s run-time checks.

The paper is organized as follows: Section C.2 introduces the target CPU architecture (ARMv7A), the architecture of the existing hypervisor and its interactions with the hosted Linux kernel, the threat model and the existing formal models; Section B.3 describes the MProsper architecture and design, it also elaborates on the additional software required to host Linux; Section B.4 describes the formal model of the monitor and formally states the top level goal: absence of code injection; Section C.6 presents the verification strategy, by summarizing the proofs that have been implemented in HOL4; Section B.6 demonstrates the overhead of MProsper through standard microbenchmarks, it also presents measures of the code and proof bases; finally, Sections D.2 and B.8 present the related work and the concluding remarks.

B.2 Background

B.2.1 The Prosper Hypervisor

The Prosper hypervisor supports the execution of an untrusted Linux guest [157] along with several trusted components. The hosted Linux is paravirtualized; both applications and kernel are executed unprivileged (in user mode) while privileged operations are delegated to the hypervisor, which is invoked via hypercalls. The physical memory region allocated to each component is statically defined. The hypervisor guarantees spatial isolation of the hosted components; a component can not directly affect (or be affected by) the content of the memory regions allocated to other components. Thus, the interactions among the hosted components are possible only via controlled communication channels, which are supervised by the hypervisor.

The Prosper hypervisor and the MProsper monitor target the ARMv7-A architecture, which is the most widely adopted instruction set architecture in mobile computing. In ARMv7-A, the virtual memory is configured via page tables that reside in physical memory. The architecture provides two levels of page tables, in the following called L1s and L2s. These tables represent the configuration of the Memory Management Unit (MMU) and define the access permissions to the virtual memory. As is common among modern architectures, the entries of ARMv7 page tables support the NX (No eXecute) attribute: an instruction can be executed only if it is fetched from a virtual memory area whose NX bit is not set. Therefore, the system executable code is a subset of the content of the physical blocks that have at least an executable virtual mapping.

To isolate the components, the hypervisor takes control of the MMU and configures the pagetables so that no illicit access is possible. This MMU configuration can not be static; the hosted Linux must be able to reconfigure the layout of its own

request r	DMMU behavior
$switch(bl)$	makes block bl the active page table
$free_{L1}(bl)$ and $free_{L2}(bl)$	frees block bl , by setting its type to <i>data</i>
$unmap_{L1}(bl, idx)$, $unmap_{L2}(bl, idx)$	unmaps entry idx of the page table stored in block bl
$link_{L1}(bl, idx, bl')$	maps entry idx of block bl to point the L2 stored in bl'
$map_{L2}(bl, idx, bl', ex, wt, rd)$ and $map_{L1}(bl, idx, bl', ex, wt, rd)$	map entry idx of block bl to point to block bl' and granting rights ex, wt, rd to user mode
$create_{L2}(bl)$ and $create_{L1}(bl)$	makes block bl a potential $L2/L1$, by setting its type to $L2/L1$

Table B.1: DMMU API

memory (and the memory of the user programs). For this reason the hypervisor virtualizes the memory subsystem. This virtualization consists of a set of APIs that enable Linux to request the creation/deletion/modification of a page table and to switch the one currently used by the MMU.

Similarly to Xen [26], the virtualization of the memory subsystem is accomplished by direct paging. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Once the page tables are activated, the hypervisor must guarantee that further updates are possible only via the virtualization API.

The physical memory is fragmented into blocks of 4 KB. Thus, a 32-bit architecture has 2^{20} physical blocks. We assign a type to each physical block, that can be: *data*: the block can be written by the guest, $L1$: contains part of an $L1$ page table and should not be writable by the guest, $L2$: contains four $L2$ page tables and should not be writable by the guest. We call the $L1$ and $L2$ blocks “potential” page tables, since the hypervisor allows to select only these memory areas to be used as page tables by the MMU.

Table B.1 summarizes the APIs that manipulate the page tables. The set of these functions is called DMMU. Each function validates the page type, guaranteeing that page tables are write-protected. A naive run-time check of the page-type policy is not efficient, since it requires to re-validate the $L1$ page table whenever the *switch* hypercall is invoked. To efficiently enforce that only blocks typed *data* can be written by the guest the hypervisor maintains a reference counter, which tracks for each block the sum of descriptors providing access in user mode to the block. The intuition is that a hypercall can change the type of a physical block (e.g. allocate or free a page table) only if the corresponding reference counter is zero.

A high level view of the hypervisor architecture is depicted in Fig. B.1. The

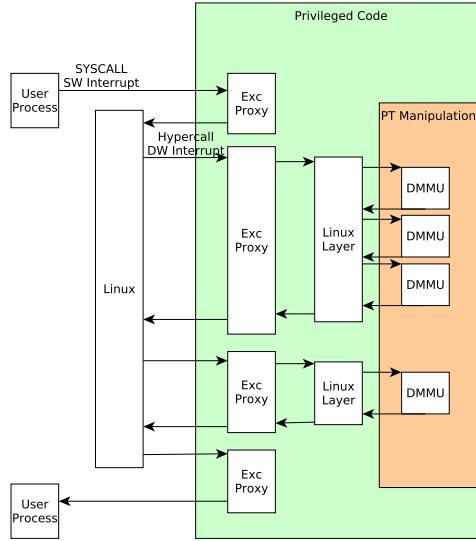


Figure B.1: Hypervisor Architecture

hypervisor is the only component that is executed in privileged mode. It logically consists of three layers: (i) an interface layer (e.g. the exception handlers) that is independent from the hosted software, (ii) a Linux specific layer and (iii) a critical core (i.e. the DMMU), which is the only component that manipulates the sensible resources (i.e. the page tables). Fig. B.1 demonstrates the behavior of the system when a user process in the Linux guest spawns a new process.

This design has two main benefits: (i) the critical part of the hypervisor is small and does not depend on the hosted software and (ii) the Linux-specific layer enriches the expressiveness of the hypercalls, thus reducing the number of context switches between the hypervisor and the Linux kernel. From a verification point of view, to guarantee security of the complete system it is not necessary to verify functional correctness of the Linux layer; it suffices to verify that this layer never changes directly the sensitive resources and that its execution does not depend on the sensitive data. These tasks can be accomplished using sand-boxing techniques [188] or tools for information flow analysis [24].

B.2.2 The Attack Model

The Linux guest is not trusted, thus we take into account an attacker that has complete control of the partition that hosts Linux. The attacker can force user programs and the Linux kernel to follow arbitrary flows and to use arbitrary data. The attacker can invoke the hypervisor, including the DMMU API, through software interrupts and exceptions. Other transitions into privileged memory are prevented

by the hypervisor. The goal of the attacker is “code injection”, for example using a buffer overflow to inject malicious executable code. This attack is normally performed by a malicious software that is able to write code into a data storage area of another process, and then cause this code to be executed.

In this paper we exemplify our monitor infrastructure using code signing. Signing the system code is a widely used approach to confirm the software author and guarantee (computationally speaking) that the code has not been altered or corrupted, by use of a cryptographic hash. Many existing code signing systems rely on a public key infrastructure (PKI) to provide both code authenticity and integrity. Here we use code signing to define integrity of the system code: integrity of an executable physical block stands for the block having a valid signature. Similarly, the integrity of the system code depends on the integrity of all executable physical blocks. The valid signatures are assumed to be known by the runtime monitor. We refer to this information as the “golden image” (GI) and it is held by the monitor.

In order to make injected code detectable, we also assume that the attacker is computationally bound; it can not modify the injected code to make its signature compliant with the golden image. We stress that our goal is not to demonstrate the security properties of a specific signature scheme. In fact the monitor can be equipped with an arbitrary signature mechanism and the signature mechanism itself is just one of the possible approaches that can be used to check integrity of the system code. For this reason we do not elaborate further on the computational power of the attacker.

B.2.3 Formal Model of the Hypervisor

Our formal model is built on top of the existing HOL4 model for ARMv7 [82]. This has been extended with a detailed formalization of the ARMv7 MMU, so that every memory access uses virtual addresses and respects the constraints imposed by the page tables.

An ARMv7 state is a record $\sigma = \langle \text{regs}, \text{coregs}, \text{mem} \rangle \in \Sigma$, where regs , coregs and mem , respectively, represent the registers, coprocessors and memory. In the state σ , the function $\text{mode}(\sigma)$ determines the current privilege execution mode, which can be either $PL0$ (user mode, used by Linux and the monitor) or $PL1$ (privileged mode, used by the hypervisor).

The system behavior is modeled by a state transition relation $\xrightarrow{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, representing the complete execution of a single ARM instruction. Non-privileged transitions ($\sigma \xrightarrow{PL0} \sigma'$) start and end in $PL0$ states. All the other transitions ($\sigma \xrightarrow{PL1} \sigma'$) involve at least one state in privileged level. A transition from $PL0$ to $PL1$ is done by raising an exception, that can be caused by software interrupts, illegitimate memory accesses, and hardware interrupts.

The transition relation queries the MMU to translate the virtual addresses and to check the access permissions. The MMU is represented by the function $\text{mmu}(\sigma, PL, va, accreq) \rightarrow pa \cup \{\perp\}$: it takes the state σ , a privilege level PL , a

virtual address $va \in 2^{32}$ and the requested access right $accreq \in \{rd, wt, ex\}$, for *readable*, *writable*, and *executable* in non-privileged respectively, and returns either the corresponding physical address $pa \in 2^{32}$ (if the access is granted) or a fault (\perp).

In [157] we show that a system hosting the hypervisor resembles the following abstract model. A system state is modeled by a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ and an abstract hypervisor state h , of the form $\langle \tau, \rho_{ex}, \rho_{wt} \rangle$. Let $bl \in 2^{20}$ be the index of a physical block and $t \in \{D, L1, L2\}$, $\tau \vdash bl : t$ tracks the type of the block and $\rho_{ex}(bl), \rho_{wt}(bl) \in 2^{30}$ track the reference counters: the number of page tables entries (i.e. entries of physical blocks typed either $L1$ or $L2$) that map to the physical block bl and are executable or writable respectively.

The transition relation for this model is $\langle \sigma, h \rangle \xrightarrow{\alpha} \langle \sigma', h' \rangle$, where $\alpha \in \{0, 1\}$, and is defined by the following inference rules:

- if $\sigma \xrightarrow{PL0} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$; instructions executed in non-privileged mode that do not raise exceptions behave equivalently to the standard ARMv7 semantics and do not affect the abstract hypervisor state.
- if $\sigma \xrightarrow{PL1} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{1} H_r(\langle \sigma', h \rangle)$, where $r = req(\sigma')$; whenever an exception is raised, the hypervisor is invoked through a hypercall, and the reached state is resulting from the execution of the handler H_r

Here, req is a function that models the hypercall calling conventions; the target hypercall is identified by the first register of σ , and the other registers provide the hypercall's arguments. The handlers H_r formally model the behavior of the memory virtualization APIs of the hypervisor (see Table B.1).

Intuitively, guaranteeing spatial isolation means confining the guest to manage a part of the physical memory available for the guest uses. In our setting, this part is determined statically and identified by the predicate $G_m(bl)$, which holds if the physical block bl is part of the physical memory assigned to the guest partition. Clearly, no security property can be guaranteed if the system starts from a non-consistent state; for example the guest can not be allowed to change the MMU behavior by directly writing the page tables. For this reason we introduce a system invariant $I_H(\langle \sigma, h \rangle)$ that is used to constrain the set of consistent initial states. Then the hypervisor guarantees that the invariant is preserved by every transition:

Proposition 1. *Let $I_H(\langle \sigma, h \rangle)$. If $\langle \sigma, h \rangle \xrightarrow{i} \langle \sigma', h' \rangle$ then $I_H(\langle \sigma', h' \rangle)$.*

We use the function $content : \Sigma \times 2^{20} \rightarrow 2^{4096*8}$ that returns the content of a physical block in a system state as a value of 4 KB. Proposition 2 summarizes some of the security properties verified in [157]: the untrusted guest can not directly change (1) the memory allocated to the other components, (2) physical blocks that contain potential page tables, (3) physical blocks whose writable reference counter is zero and (4) the behavior of the MMU.

Proposition 2. *Let $I_H(\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}) \rangle)$. If $\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}) \rangle \xrightarrow{0} \langle \sigma', h' \rangle$ then:*

1. For each DMMU hypercall invoked by a guest, the hypervisor forwards the hypercall’s request to the monitor.
2. The monitor validates the request based on its validation mechanism.
3. The monitor reports to the hypervisor the result of the hypercall validation.

Figure B.2: The interaction protocol between the Prosper hypervisor and the monitor

- For every bl such that $\neg G_m(bl)$ then $content(bl, \sigma) = content(bl, \sigma')$
- For every bl such that $\tau(bl) \neq data$ then $content(bl, \sigma) = content(bl, \sigma')$
- For every bl if $content(bl, \sigma) \neq content(bl, \sigma')$ then $\rho_{wt}(bl) > 0$
- For every va, PL, acc we have $mmu(\sigma, va, PL, acc) = mmu(\sigma', va, PL, acc)$

B.3 Design

We configured the hypervisor to support the interaction protocol of Figure B.2; the monitor mediates accesses to the DMMU layer. Since the hypervisor supervises the changes of the page tables the monitor is able to intercept all modifications to the memory layout. This makes the monitor able to know if a physical block is writable: This is the case if there exists at least one virtual mapping pointing to the block with a guest writable access permission. Similarly it is possible to know if a physical block is executable. Note that the identification of the executable code (also called “working set”) does not rely on any information provided by the untrusted guest. Instead, the monitor only depends on HW information, which can not be tampered by an attacker.

The first policy enforced by the monitor is code signature: Whenever Linux requests to change a page table (i.e. causing to change the domain of the working set) the monitor (i) identifies the physical blocks that can be made executable by the request, (ii) computes the block signature and (iii) compares the result with the content of the golden image. This policy is sufficient to prevent code injection that are caused by changes of the memory layout setting, due to the hypervisor forwarding to the monitor all requests to change the page tables.

However, this policy is not sufficient to guarantee integrity of the working set. In fact, operations that modify the content of a physical block that is executable can violate the integrity of the executable code. These operations cannot be intercepted by the monitor, since they are not supposed to raise any hypercall. In fact, a simple write operation in a block typed *data* does not require the hypervisor intermediation since no modification of the memory layout is introduced. To prevent code injections performed by writing malicious code in an executable area of the memory, the

monitor enforces the executable space protection policy $W \oplus X$, preventing physical blocks from being simultaneously writable and executable. As for the hypervisor, a naive run-time check of the executable space protection is not efficient. Instead, we reuse the hypervisor reference counters: we accept a hypercall that makes a block executable (writable) only if the writable (executable) reference counter of the block is zero.

An additional complication comes from the Linux architecture. An unmodified Linux kernel will not survive the policies enforced by the monitor, thus its execution will inevitably fail. For example, when a user process is running there are at least two virtual memory regions that are mapped to the same physical memory where the process executable resides: (i) the user “text segment” and (ii) the “kernel space” (which is an injective map to the whole physical memory). When the process is created, Linux requests to set the text segment as executable and non writable. However, Linux does not revoke its right to write inside this memory area using its kernel space. This setting is not accepted by the monitor, since it violates $X \oplus W$, thus making it impossible to execute a user process.

Instead of adapting a specific Linux kernel we decided to implement a small emulation layer that has two functionalities:

- It proxies all requests from the Linux layer to the monitor. If the emulator receives a request that can be rejected by the monitor (e.g. a request setting as writable a memory region that is currently executable) then the emulator (i) downgrades the access rights of the request (e.g. setting them as non writable) and (ii) stores the information about the suspended right in a private table.
- It proxies all data and prefetch aborts. The monitor looks up in the private table to identify if the abort is due to an access right that has been previously downgraded by the emulator. In this case the monitor attempts (i) to downgrade the existing mapping that conflicts with the suspended access right and (ii) to re-enable the suspended access right.

Note that a malfunction of the emulation layer does not affect the security of the monitor. Namely, we do not care if the emulation layer is functionally correct, but only that it does not access sensible resources directly.

Fig. B.3 depicts the architecture of MPProsper. Both the runtime monitor and the emulator are deployed as two guests of the Prosper hypervisor. The Linux layer prepares a list of requests in a buffer shared with the emulation guest. After the Linux layer returns, the hypervisor activates the emulation guest, which manipulates the requests (or adds new ones) as discussed before. Then the hypervisor iteratively asks the monitor to validate one of the pending requests and upon success it commits the request by invoking the corresponding DMMU function.

Using a dedicated guest on top of the hypervisor permits to decouple the enforcement of the security policies from the other hypervisor functionalities, thus keeping the trusted computing base minimal. Moreover, having the security policy

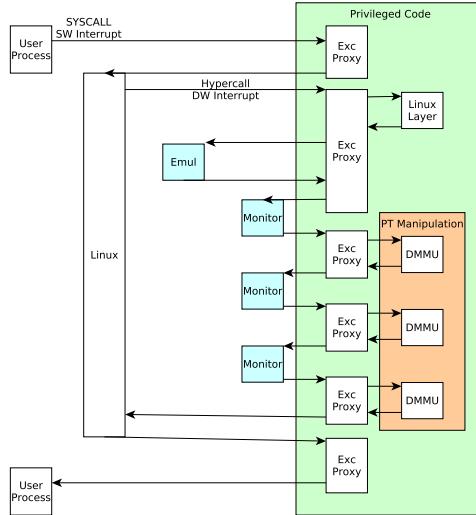


Figure B.3: MProsper’s Architecture

wrapped inside a guest supports both the tamper-resistance and the trustworthiness of the monitor. In fact, the monitor can take advantage from the isolation properties provided by the hypervisor. This avoids malicious interferences coming from the other guests (for example from a process of an OS running on a different partition of the same machine). Finally, decoupling the run-time security policy from the other functionalities of the hypervisor makes the formal specification and verification of the monitor more affordable.

B.4 Formal Model of MProsper

The formal model of the system (i.e. consisting of the hypervisor, the monitor and the untrusted Linux) is built on top of the models presented in Section B.2.3. Here we leave unspecified the algorithm used to sign and check signatures, so that our results can be used for different intrusion detection mechanisms. The golden image GI is a finite set of signatures $\{s_1, \dots, s_n\}$, where the signatures are selected from a domain S . We assume the existence of a function $sig : 2^{4096*8} \rightarrow S$ that computes the signature of the content of a block. The system behavior is modeled by the following rules:

1.
$$\frac{\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h' \rangle}{\langle \sigma, h, GI \rangle \xrightarrow{0} \langle \sigma', h', GI \rangle}$$
2.
$$\frac{\langle \sigma, h \rangle \xrightarrow{1} \langle \sigma', h' \rangle \text{ validate}(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \xrightarrow{1} \langle \sigma', h', GI \rangle}$$

$$3. \quad \frac{\langle \sigma, h \rangle \xrightarrow{1} \langle \sigma', h' \rangle \ \neg validate(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)}{\langle \sigma, h, GI \rangle \xrightarrow{1} \epsilon(\langle \sigma, h, GI \rangle)}$$

User mode transitions (e.g. Linux activities) require neither hypervisor nor monitor intermediation. Proposition 2 justifies the fact that, by construction, the transitions executed by the untrusted component can not affect the monitor state; (i) the golden image is constant and (ii) the monitor code can be statically identified and abstractly modeled. Executions in privileged mode require monitor intermediation. If the monitor validates the request, then the standard behavior of the hypervisor is executed. Otherwise the hypervisor performs a special operation to reject the request, by reaching the state that is returned by a function ϵ . Hereafter, the function ϵ is assumed to be the identity. Alternatively, ϵ can transform the state so that the requestor is informed about the rejected operation, by updating the user registers according to the desired calling convention.

The function $validate(req(\langle \sigma, h \rangle), \langle \sigma, h, GI \rangle)$ represents the validation mechanism of the monitor, which checks at run-time possible violations of the security policies. In Table B.2 we briefly summarize the policies for the different access requests. Here, PT is a function that yields the list of mappings granted by a page table, where each mapping is a tuple (vb, pb, wt, ex) containing the virtual block mapped (vb), the pointed physical block (pb) and the unprivileged rights to execute (ex) and write (wt). The rules in Table B.2 are deliberately more abstract than the ones modeled in HOL4 and are used to intuitively present the behavior of the monitor. For example, the function PT is part of the hardware model and is not explicitly used by the monitor code, it is instead more similar to an iterative program. This makes our verification more difficult, but it also makes the monitor model as near as possible to the actual implementation, enabling further verification efforts that can establish correctness of the implementation.

Note that the monitor always checks that a mapping is not writable and executable simultaneously. Furthermore, if a mapping grants a writable access then the executable reference counter of the pointed physical block must be zero, guaranteeing that this mapping does not conflict (according with the executable space protection policy) with any other allocated page table. Similarly, if a mapping grants an executable access, then the writable reference counter of the pointed block must be zero.

To formalize the top goal of our verification we introduce some auxiliary notations. The working set identifies the physical blocks that host executable binaries and their corresponding content.

Definition Let σ be a machine state. The working set of σ is defined as

$$WS(\sigma) = \{ \langle bl, content(bl, \sigma) \rangle \mid \exists pa, va. mmu(\sigma, PL0, va, ex) = pa \wedge pa \in bl \}$$

By using a code signing approach, we say that the integrity of a physical block is satisfied if the signature of the block's content belongs to the golden image.

request r	$validate(r, \langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI \rangle)$ holds iff
$switch(bl)$	always
$free_{L1}(bl)$ and $free_{L2}(bl)$	always
$unmap_{L1}(bl, idx)$, $unmap_{L2}(bl, idx)$, and $link_{L1}(bl, idx, bl')$	$\rho_{ex}(bl) = 0$
$map_{L2}(bl, idx, bl', ex, wt, rd)$ and $map_{L1}(bl, idx, bl', ex, wt, rd)$	$sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl')$ and $sound_S(ex, bl', \sigma, GI) \wedge \rho_{ex}(bl) = 0$
$create_{L2}(bl)$ and $create_{L1}(bl)$	$\forall(vb, pb, wt, ex) \in PT(content(bl, \sigma))$. $sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, pb)$ and $sound_S(ex, pb, \sigma, GI)$ $\forall(vb', pb', wt', ex') \in PT(content(bl, \sigma))$. $no-conflict(vb, pb, wt, ex)(vb', pb', wt', ex')$

where

$$sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl) = \left(\begin{array}{l} (ex \Rightarrow \neg wt \wedge \rho_{wt}(bl) = 0) \wedge \\ (wt \Rightarrow \neg ex \wedge \rho_{ex}(bl) = 0) \end{array} \right)$$

$$sound_S(ex, bl, \sigma, GI) = (ex \Rightarrow integrity(GI, bl, content(bl, \sigma)))$$

$$no-conflict(vb, pb, wt, ex)(vb', pb', wt', ex') = \left(\begin{array}{l} (vb \neq vb' \wedge pb = pb') \Rightarrow \\ (ex \Rightarrow \neg wt' \wedge wt \Rightarrow \neg ex') \end{array} \right)$$

Table B.2: Security policies for the available access requests

Definition Let $cnt \in 2^{4096*8}$ be the 4KB content of a physical block bl and GI be the golden image. Then $integrity(GI, bl, cnt)$ if, and only if, $sig(bl, cnt) \in GI$

Notice that our security property can be refined to fit different anti-intrusion mechanisms. For example, $integrity(GI, bl, cnt)$ can be instantiated with the execution of an anti-virus scanner.

The system state is free of malicious code injection if the signature check is satisfied for the whole executable code. That is:

Definition Let σ be a machine state, bl be a physical block and GI be the golden image. Then $integrity(GI, \sigma)$ if, only only if, for all $\langle bl, cnt \rangle \in WS(\sigma)$, $integrity(GI, bl, cnt)$

Finally, we present our top level proof goal: No code injection can succeed.

Proposition 3. If $\langle \sigma, h, GI \rangle$ is a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI \rangle$ then $integrity(GI, \sigma)$

B.5 Verification Strategy

Our verification strategy consists of introducing a state invariant $I(s)$ that is preserved by any possible transition and demonstrating that the invariant guarantees the desired security properties.

Definition $I(\sigma, (\tau, \rho_{wt}, \rho_{ex}), GI)$ holds if

$$\begin{aligned} & I_H(\sigma, (\tau, \rho_{wt}, \rho_{ex})) \wedge \\ & \forall bl . (\neg(\tau(bl) = data)) \Rightarrow \forall (vb, pb, wt, ex) \in PT(content(bl, \sigma)). \\ & \quad sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, pb) \wedge sound_S(ex, pb, \sigma, GI) \end{aligned}$$

Clearly, the soundness of the monitor depends on the soundness of the hypervisor, thus I requires that the hypervisor's invariant I_H holds. Notice that the invariant constrains not only the page tables currently in use, but it constrains all potential page tables, which are all the blocks that have type different from $data$. This allows to speed up the context switch, since the guest simply re-activates a page table that has been previously validated. Technically, the invariant guarantees protection of the memory that can be potentially executable and the correctness of the corresponding signatures.

We verified independently that the invariant is preserved by unprivileged transitions (Theorem B.5.1) and by privileged transitions (Theorem B.5.2). Moreover, Lemma B.5.1 demonstrates that the monitor invariant guarantees there is no malicious content in the executable memory.

Lemma B.5.1. *If $I(\langle\sigma, (\tau, \rho_{wt}, \rho_{ex}), GI\rangle)$ then $integrity(GI, \sigma)$.*

Proof. The proof is straightforward, following from $sound_S$ of every block that can be executable according with an arbitrary potential page table. \square

Theorem B.5.1 demonstrates that the invariant is preserved by instructions executed by the untrusted Linux. This depends on Lemma B.5.2, which shows that the invariant forbids user transitions to change the content of the memory that is executable.

Lemma B.5.2. *Let $\langle\sigma, (\tau, \rho_{wt}, \rho_{ex}), GI\rangle \xrightarrow{0} \langle\sigma', h', GI'\rangle$ and $I(\langle\sigma, h, GI\rangle)$ then*

$$\forall bl . (\neg(\tau(bl) = data)) \Rightarrow \left(\begin{array}{l} PT(content(bl, \sigma')) = PT(content(bl, \sigma)) \wedge \\ \forall (vb, pb, wt, ex) \in PT(content(bl, \sigma')) . \\ (ex \Rightarrow content(pb, \sigma) = content(pb, \sigma')) \end{array} \right)$$

Proof. Proof is straightforward and we split it in two parts. First we show that page tables remain constant after user transitions and then we prove that executable block cannot be changed by the user. From Proposition 2 we know since the hypervisor invariant holds in the state $\langle\sigma, h, GI\rangle$, user transitions are not allowed to change the page tables. Thus the user transitions preserve mappings of all page tables in the memory.

By the monitor's invariant we know that in $\langle \sigma, h, GI \rangle$ all the mappings in page tables comply with the policy $sound_{W \oplus X}$. That is, if a mapping in a page table grants executable permissions on block to the user, the block must be write protected against unprivileged accesses. Moreover, we know that page tables remain the same after an unprivileged transition and the user cannot change his access permission. This proves that the content of executable block is not modifiable by unprivileged transitions. \square

Theorem B.5.1. *If $\langle \sigma, h, GI \rangle \xrightarrow{0} \langle \sigma', h', GI' \rangle$ and $I(\langle \sigma, h, GI \rangle)$ then $I(\langle \sigma', h', GI' \rangle)$.*

Proof. From the inference rules we know that $h' = h$, $GI' = GI$ and that the system without the monitor behaves as $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$. Thus, Proposition 1 can be used to guarantee that the hypervisor invariant is preserved ($I_H(\sigma', h')$).

If the second part of the invariant is violated then there must exist a mapping in one (hereafter bl) of the allocated page tables that is compliant with the executable space protection policy in σ and violates the policy in σ' . Namely, $content(bl, \sigma')$ must be different from $content(bl, \sigma)$. This contradicts Proposition 2, since the type of the changed block is not data ($\tau(bl) \neq data$). \square

Finally we must demonstrate that every potentially executable block contains a sound binary. Lemma B.5.2 guarantees that the blocks that are potentially executable are the same in σ and σ' and that the content of these blocks is unchanged. Thus is sufficient to use the invariant $I(\sigma, h, GI)$, to demonstrate that the signatures of all executable blocks are correct. \square

To demonstrate the functional correctness of the monitor (Theorem B.5.2 i.e. that the invariant is preserved by privileged transitions) we introduce two auxiliary lemmas: Lemma B.5.3 shows that the monitor correctly checks the signature of pages that are made executable. Lemma B.5.4 expresses that executable space protection is preserved for all hypervisor data changes, as long as a block whose reference counter (e.g. writable; ρ'_{wt}) becomes non zero has the other reference counter (e.g. executable; ρ_{ex}) zero.

Lemma B.5.3. *If $\langle \sigma, h, GI \rangle \xrightarrow{1} \langle \sigma', (\tau', \rho'_{wt}, \rho'_{ex}), GI' \rangle$ and $I(\langle \sigma, h, GI \rangle)$ then for all bl , $\tau'(bl) \neq data \Rightarrow \forall (vb', pb', wt, ex) \in PT(content(bl, \sigma')). sound_S(ex, pb', \sigma', GI)$.*

Proof. Proof is done by case analysis on the type of the performed operation and its the validity. If the requested operation is not valid then the proof is trivial, since ϵ is the identity function and it yields the same state. Therefore, $sound_S$ holds.

However, if the request is validated by the monitor and it is committed by the hypervisor we prove Lemma B.5.3 by doing case analysis on the type of the performed operation. Let assume that the requested operation is map_{L2} .

- (i) If the requested mapping does not set the executable flag ex , the working set would not be changed by committing the request $WS(\sigma) = WS(\sigma')$ and the predicate $sound_S$ holds trivially.

- (ii) If the requested mapping sets the executable flag ex (i.e. the mapping grants an executable access to a physical block pb), from validity of the request we know that $sound_S(ex, pb, \sigma, GI)$. Thus adding this mapping preserves the soundness of state and the lemma holds in this case as well.

Proof for the other operations can be done similarly. \square

Lemma B.5.4. *If $\langle \sigma, h, GI \rangle \xrightarrow{1} \langle \sigma', (\tau', \rho'_{wt}, \rho'_{ex}), GI' \rangle$ and assuming that:*

- (i) $I(\langle \sigma, (\tau, \rho_{wt}, \rho_{ex}), GI \rangle)$,
- (ii) $\forall bl. (\rho_{ex}(bl) = 0 \wedge \rho'_{ex}(bl) > 0) \Rightarrow (\rho_{wt}(bl) = 0)$, and
- (iii) $\forall bl. (\rho_{wt}(bl) = 0 \wedge \rho'_{wt}(bl) > 0) \Rightarrow (\rho_{ex}(bl) = 0)$.

For all blocks bl , if $sound_{W \oplus X}(wt, ex, \rho_{wt}, \rho_{ex}, bl)$ then $sound_{W \oplus X}(wt, ex, \rho'_{wt}, \rho'_{ex}, bl)$

Proof. Since in the initial state the invariant holds, we know that the state $\langle \sigma, h, GI \rangle$ complies with the policy $W \oplus X$. Thus for each block bl only one of its counters $\rho_{ex}(bl)$ and $\rho_{wt}(bl)$ can be greater than zero. This implies that only one of the assumptions (ii), (iii) is true. Let assume that in $\langle \sigma, h, GI \rangle$ the block bl is executable, therefore $\rho_{wt}(bl) = 0$. If in $\langle \sigma, h', GI \rangle$, the block bl is still executable, its writable permission has not been changed and $sound_{W \oplus X}(wt, ex, \rho'_{wt}, \rho'_{ex}, bl)$ holds. However, its permissions has been changed in $\langle \sigma, h', GI \rangle$ it has to be happened according to the assumption (iii), this means that $sound_{W \oplus X}(wt, ex, \rho'_{wt}, \rho'_{ex}, bl)$ will be hold true. \square

Theorem B.5.2. *If $\langle \sigma, h, GI \rangle \xrightarrow{1} \langle \sigma', h', GI' \rangle$ and $I(\langle \sigma, h, GI \rangle)$ then $I(\langle \sigma', h', GI' \rangle)$.*

Proof. When the request is not validated ($\neg validate$) then the proof is trivial, since ϵ is the identity function.

If the request is validated by the monitor and committed by the hypervisor, then the inference rules guarantee that $GI' = GI$ and that the system without the monitor behaves as $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$. Thus, Proposition 1 can be used to guarantee that the hypervisor invariant is preserved ($I_H(\sigma', h')$). Moreover, Lemma B.5.3 demonstrates that the $sound_S$ part of the invariant holds.

The proof of the second part (the executable space protection) of the invariant is the most challenging task of this formal verification. This basically establishes the functional correctness of the monitor and that its run-time policies are strong enough to preserve the invariant (i.e. they enforce protection of the potentially executable space). Practically speaking, the proof consists of several cases: one for each possible request. The structure of the proof for each case is similar. For example, for $r = map_{L2}(bl, idx, bl', ex, wt, rd)$, we (i) prove that the hypervisor (modeled by the function H_r) only changes entry idx of the page table stored in block bl (that is, all other blocks that are not typed $data$ are unchanged), (ii) we show that only the counters of physical block bl' are changed, and (iii) we

establish the hypothesis of Lemma B.5.4. This enables us to infer $sound_{W \oplus X}$ for the unchanged blocks and to reduce the proof to only check the correctness of the entry idx of the page table in the block bl . \square

Finally, Theorem B.5.3 composes our results, demonstrating that no code injection can succeed.

Theorem B.5.3. *Let $\langle \sigma, h, GI \rangle$ be a state reachable from the initial state of the system $\langle \sigma_0, h_0, GI_0 \rangle$ and $I(\langle \sigma_0, h_0, GI_0 \rangle)$, then $integrity(GI, \sigma)$ holds*

Proof. Theorems B.5.1 and B.5.2 directly show that the invariant is preserved for an arbitrary trace. Then, Lemma B.5.1 demonstrates that every reachable state is free of malicious code injection. \square

B.6 Evaluation

The verification has been performed using the HOL4 interactive theorem prover. The specification of the high level model of the monitor adds 710 lines of HOL4 to the existing model of the hypervisor. This specification is intentionally low level and does not depend on any high level theory of HOL4. This increased the difficulty of the proof (e.g., it musts handle finite arithmetic overflows), that consists of 4400 lines of HOL4. However, the low level of abstraction allowed us to directly transfer the model to a practical implementation and to identify several bugs of the original design. For example, the original policy for $link_{L1}(bl, idx, bl')$ did not contain the condition $\rho_{ex}(bl) = 0$, allowing to violate the integrity of the working set if a block is used to store an L1 page table and is itself executable.

The monitor code consists of 720 lines of C and the emulator consists of additional 950 lines of code. Finally, 100 lines have been added to the hypervisor to support the needed interactions among the hosted components.

We used LMBench to measure the overhead introduced on user processes hosted by Linux. We focused on the benchmarks “fork”, “exec” and “shell”, since they require the creation of new processes and thus represent the monitors worst case scenario. As macro-benchmark, we measured in-memory compression of two data streams. The benchmarks have been executed using Qemu to emulate a Beagleboard-Mx. Since we are not interested in evaluating a specific signature scheme, we computed the signature of each physical block as the xor of the contained words, allowing us to focus on the overhead introduced by the monitor’s infrastructure. Table B.3 reports the benchmarks for different prototypes of the monitor, thus enabling to compare the overhead introduced by different design choices. “No monitor” is the base configuration, where neither the monitor or the emulation layer are enabled. In “P Emu” the emulation layer is enabled and deployed as component of the hypervisor. This benchmark is used to measure the overhead introduced by this layer, which can be potentially removed at the cost of modifying the Linux

Benchmark	fork	exec	shell	tar -czvf 1.2 KB/12.8 MB	
No monitor	0.010	0.010	0.042	0.05	20.95
P Emu	0.011	0.011	0.048	0.09	21.05
P Emu + P Mon	0.013	0.013	0.054	0.10	21.02
P Emu + U Mon	0.017	0.017	0.067	0.11	20.98

Table B.3: Qemu benchmarks [in second]

kernel. In “P Emu + P Mon” both the monitor and the emulation layer are deployed as privileged software inside the hypervisor. Finally, in “P Emu + U Mon” the monitor is executed as unprivileged guest.

B.7 Related Work

Since a comprehensive verification of commodity SW is not possible, it is necessary to architect systems so that the trusted computing base for the desired properties is small enough to be verified, and that the untrusted code cannot affect the security properties. Specialized HW (e.g. TrustZone and TPM) has been proposed to support this approach and has been used to implement secure storage and attestation. The availability of platforms like hypervisors and microkernels extended the adoption of this approach to use cases that go beyond the ones that can be handled using static HW based solutions.

For example, in [131] the authors use the seL4 microkernel to implement a secure access controller (SAC) with the purpose of connecting one front-end terminal to either of two back-end networks one at a time. The authors delegate the complex (and non security-critical) functionalities (e.g. IP/TCP routing, WEB front-end) to untrusted Linuxes, which are isolated by the microkernel from a small and trusted router manager. The authors describe how the system’s information flow properties can be verified disregarding the behavior of the untrusted Linuxes.

Here, we used the trustworthy components to help the insecure Linux to restrict its own attack surface; i.e. to prevent binary code injection. Practically, our proposal uses Virtual Machine Introspection (VMI), which has been first introduced by Garfinkel *et al.* [20] and Chen *et al.* [48]. Similarly to MProsper, other proposals (including Livewire [20], VMWatcher [123] and Patagonix [143]) use VMI, code signing and executable space protection to prevent binary code injection in commodity OSs. However, all existing proposals rely on untrusted hypervisors and their designs have not been subject of formal verification.

Among others non trustworthy VMIs, hytux [136], SecVisor [189] and NICKLE [175] focus on protecting integrity of the sole guest kernel. SecVisor establishes a trusted channel with the user, which must manually confirm all changes to the kernel. NICKLE uses a *shadow memory* to keep copy of authenticated modules and guarantees that any instruction fetch by the kernel is routed to this memory.

OpenBSD 3.3 has been one of the first OS enforcing executable space protection ($W \oplus X$). Similarly, Linux (with the PaX and Exec Shield patches), NetBSD

and Microsoft’s OSs (using Data Execution Prevention (DEP)) enforce the same policy. However, we argue that due to the size of the modern kernels, trustworthy executable space protection can not be achieved without the external support of a trusted computing base. In fact, an attacker targeting the kernel can circumvent the protection mechanism, for example using *return-oriented programming* [191]. The importance of enforcing executable space protection from a privileged point of view (i.e. by VMI) is also exemplified by [141]. Here, the authors used model checking techniques to identify several misbehaviors of the Linux kernel that violate the desired property.

B.8 Concluding Remarks

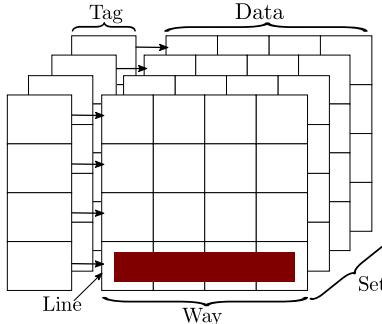
We presented a trustworthy code injection prevention system for Linux on embedded devices. The monitor’s trustworthiness is based on two main principles (i) the trustworthy hypervisor guarantees the monitor’s tamper resistance and that all memory operations that modify the memory layout are mediated, (ii) the formal verification of design demonstrates that the top security goal is guaranteed by the run-time checks executed by the monitor. These are distinguishing features of MProsper, since it is the first time that absence of binary code injection has been verified for a commodity OS.

Even if the MProsper’s formal model is not yet at the level of the actual binary code executed on the machine, this verification effort is important to validate the monitor design; in fact we were able to spot security issues that were not dependent on the specific implementation of the monitor. The high level model of the monitor is actually a state transition model of the implemented code, operating on the actual ARMv7 machine state. Thus the verified properties can be transferred to the actual implementation by using standard refinement techniques (e.g. [190]).

Our ongoing work include the development of a end-to-end secure infrastructure, where an administrator can remotely update the software of an embedded device. Moreover, we are experimenting with other run-time binary analysis techniques that go beyond code signature checking: for example an anti-virus scanner can be integrated with the monitor, enabling to intercept and stop self-decrypting malwares.

ARM Data-Cache Terminology

In the following we give a short summary of some of the terms used in the next two papers and a figure illustrating the structure of a data-cache in the ARM architecture:



Data-cache structure for the ARM architecture.

Tag: The tag is the part of an address (mostly physical address) stored in the cache which identifies the corresponding memory address associated to a line in the cache.

Line: A line in the cache contains a block of contiguous words for the memory. Each line include also a *valid flag* (which indicates if the line contains a valid data) and a *dirty flag* (which indicates if the line has been changed since it was read from memory).

Set: Memory addresses are logically partitioned into sets of lines that are congruent w.r.t. a set index; usually set index depends on either virtual addresses (then the cache is called virtually indexed) or physical addresses (then the cache is called physically indexed).

Way: The cache contains a number of ways which can hold one corresponding line for every set index.

Cache Flushing: Flushing means writing back into the memory a cache line and cleaning the dirty flag.

Eviction: Eviction means writing back into the memory a cache line and invalidating it in the cache.

Memory Coherence: Is the problem of ensuring that a value read from a memory location (using either a cacheable or an uncacheable address) is always the most recently written value to that location.

Paper C

Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures

C

Roberto Guanciale, Hamed Nemati, Mads Dam, Christoph Baumann

Abstract

Caches pose a significant challenge to formal proofs of security for code executing on application processors, as the cache access pattern of security-critical services may leak secret information. This paper reveals a novel attack vector, exposing a low-noise cache storage channel that can be exploited by adapting well-known timing channel analysis techniques. The vector can also be used to attack various types of security-critical software such as hypervisors and application security monitors. The attack vector uses virtual aliases with mismatched memory attributes and self-modifying code to misconfigure the memory system, allowing an attacker to place incoherent copies of the same physical address into the caches and observe which addresses are stored in different levels of cache. We design and implement three different attacks using the new vector on trusted services and report on the discovery of an 128-bit key from an AES encryption service running in TrustZone on Raspberry Pi 2. Moreover, we subvert the integrity properties of an ARMv7 hypervisor that was formally verified against a cache-less model. We evaluate well-known countermeasures against the new attack vector and propose a verification methodology that allows to formally prove the effectiveness of defence mechanisms on the binary code of the trusted software.

C.1 Introduction

Over the past decade huge strides have been made to realise the long-standing vision of formally verified execution platforms, including hypervisors [138, 157],

separation kernels [71, 174], and microkernels [133]. Many of these platforms have been comprehensively verified, down to machine code [133] and Instruction Set Architecture (ISA) [71] levels, and provide unprecedented security and isolation guarantees.

Caches are mostly excluded from these analyses. The verification of both seL4 [132] and the Prosper kernels [71, 157] assume that caches are invisible and ignore timing channels. The CVM framework from the Verisoft project [10] treats caches only in the context of device management [110]. For the verification of user processes and the remaining part of the kernel, caches are invisible. Similarly, the Nova [199, 203] and CertiKOS [97] microvisors do not consider caches in their formal analysis.

How much of a problem is this? It is already well understood that caches are one of the key features of modern commodity processors that make a precise analysis of, e.g., timing and/or power consumption exceedingly difficult, and that this can be exploited to mount timing-based side channels, even for kernels that have been fully verified [56]. These channels, thus, must be counteracted by model-external means, e.g., by adapting scheduling intervals [198] or cache partitioning [172, 129].

The models, however, should preferably be sound with respect to the features that *are* reflected, such as basic memory reads and writes. Unfortunately, as we delve deeper into the Instruction Set Architecture we find that this expectation is not met: Certain configurations of the system enable an attacker to exploit caches to build storage channels. Some of these channels are especially dangerous since they can be used to compromise both confidentiality and integrity of the victim, thus breaking the formally verified properties of isolation.

The principle idea to achieve this, is to break coherency of the memory system by deliberately not following the programming guidelines of an ISA. In this report we focus on two programming faults in particular:

1. Accessing the same physical address through virtual aliases with mismatched cacheability attributes.
2. Executing self-modifying code without flushing the instruction cache.

Reference manuals for popular architectures (ARM, Power, x64) commonly warn that not following such guidelines may result in unpredictable behaviour. However, since the underlying hardware is deterministic, the actual behaviour of the system in these cases is quite predictable and can be reverse-engineered by an attacker.

The first fault results in an incoherent memory configuration where cacheable and uncacheable reads may see different values for the same physical address after a preceding write using either of the virtual aliases. Thus the attacker can discover whether the physical address is allocated in a corresponding cache line. For the second fault, jumping to an address that was previously written without flushing the instruction cache may result in the execution of the old instruction, since data and instruction caches are not synchronised automatically. By carefully selecting

old and new instructions, as well as their addresses, the attacker can then deduce the status of a given instruction cache line.

Obtaining this knowledge, i.e., whether certain cache lines contain attacker data and instructions, is the basic principle behind the Prime+Probe flavor of access-driven timing channel attacks [208]. This type of attack can be adapted using the new attack vector. The main advantage of this approach is that the cache storage channels presented here are both more stealthy, less noisy, and easier to measure than timing channels. Moreover, an incoherent data cache state can be used to subvert the integrity of trusted services that depend on untrusted inputs. Breaking the memory coherency for the inputs exposes vulnerabilities that enable a malicious agent to bypass security monitors and possibly to compromise the integrity of the trusted software.

The attacks sketched above have been experimentally validated in three realistic scenarios. We report on the implementation of a prototype that extracts a 128-bit key from an AES encryption service running in TrustZone on Raspberry Pi 2. We use the same platform to implement a process that extracts the exponent of a modular exponentiation procedure executed by another process. Moreover, implementing a cache-based attack we subverted the integrity properties of an ARMv7 hypervisor that was formally verified against a cache-less model. The scenarios are also used to evaluate several existing countermeasures against cache-based attacks as well as new ones that are targeted to the alias-driven attack vector.

Finally, we propose a methodology to repair the formal analysis of the trusted software, reusing existing means as much as possible. Specifically, we show (1) how a countermeasure helps restoring integrity of a previously formally verified software and (2) how to prove the absence of cache storage side channels. This last contribution includes the adaptation of an existing tool [24] to analyse the binary code of the trusted software.

C.2 Background

Natural preys of side-channel attacks are implementations of cryptographic algorithms, as demonstrated by early works of Kocher [134] and Page [163]. In cache-driven attacks, the adversary exploits the caches to acquire knowledge about the execution of a victim and uses this knowledge to infer the victim’s internal variables. These attacks are usually classified in three groups, that differ by the means used by the attacker to gain knowledge. In “time-driven attacks” (e.g. [209]), the attacker, who is assumed to be able to trigger an encryption, measures (indirectly or directly) the execution time of the victim and uses this knowledge to estimate the number of cache misses and hits of the victim. In “trace-driven attacks” (e.g. [6, 163, 232]), the adversary has more capabilities: he can profile the cache activities during the execution of the victim and thus observe the cache effects of a particular operation performed by the victim. This highly frequent measurement can be possible due to the adversary being interleaved with the victim by the scheduler of the operating

system or because the adversary executes on a separate core and monitors a shared cache. Finally, in “access-driven attacks” (e.g. [158, 208]), the attacker determines the cache indices modified by the victim. This knowledge is obtained indirectly, by observing cache side effects of victim’s computation on the behaviour of the attacker.

In the literature, the majority of trace and access driven attacks use timing channels as the key attack vector. These vectors rely on time variations to load/store data and to fetch instructions in order to estimate the cache activities of the victim: the cache lines that are evicted, the cache misses, the cache hits, etc.

Storage channels, on the other hand, use system variables to carry information. The possible presence of these channels raises concerns, since they invalidate the results of formal verification. The attacker can use the storage channels without the support of an external measurement (e.g. current system time), so there is no external variable such as time or power consumption that can be manipulated by the victim to close the channel and whose accesses can alert the victim about malicious intents. Moreover, a storage channel can be less noisy than timing channels that are affected by scheduling, TLB misses, speculative execution, and power saving, for instance. Finally, storage channels can pose risk to the integrity of a system, since they can be used to bypass reference monitors and inject malicious data into trusted agents. Nevertheless, maybe due to the practical complexities in implementing these channels, few works in literature address cache-based storage channels.

One of the new attack vectors of this paper is based on mismatched cacheability attributes and has pitfalls other than enabling access-driven attacks. The vector opens up for Time Of Check To Time Of Use (TOCTTOU) like vulnerabilities. A trusted agent may check data stored in the cache that is not consistent with the data that is stored in the memory by a malicious software. If this data is later evicted from the cache, it can be subsequently substituted by the unchecked item placed in the main memory. This enables an attacker to bypass a reference monitor, possibly subverting the security property of formally verified software.

Watson [223] demonstrated this type of vulnerability for Linux system call wrappers. He uses concurrent memory accesses, using preemption to change the arguments to a system call in user memory after they were validated. Using non-cacheable aliases one could in the same way attack the Linux system calls that read from the caller’s memory. A further victim of such attacks is represented by run time monitors. Software that dynamically loads untrusted modules often uses Software-based Fault Isolation (SFI) [218, 195] to isolate untrusted components from the trusted ones. If an on-line SFI verifier is used (e.g. because the loaded module is the output of a just-in-time compiler), then caches can be used to mislead the verifier to accept stale data. This enables malicious components to break the SFI assumptions and thus the desired isolation.

In this paper we focus on scenarios where the victim and the attacker are hosted on the same system. An instance of such scenarios consists of a malicious user process that attempts to compromise either another user process, a run-time monitor or the operating system itself. In a cloud environment, the attacker can be a

(possibly compromised) complete operating system and the victim is either a collocated guest, a virtual machine introspector or the underlying hypervisor. Further instances of such scenario are systems that use specialised hardware to isolate security critical components from untrusted operating systems. For example, some ARM processors implement TrustZone [1]. This mechanism can be used to isolate and protect the system components that implement remote attestation, trusted anchoring or virtual private networks (VPN). In this case, the attacker is either a compromised operating system kernel or an untrusted user process threatening a TrustZone application.

C.3 The New Attack Vectors: Cache Storage Channels

Even if it is highly desirable that the presence of caches is transparent to program behaviour, this is usually not the case unless the system configuration satisfies some architecture-specific constraints. Memory mapped devices provide a trivial example: If the address representing the output register of a memory mapped UART is cacheable, the output of a program is never visible on the serial cable, since the output characters are overwritten in the cache instead of being sent to the physical device. These behaviours, which occur due to misconfigurations of the system, can raise to security threats.

To better understand the mechanisms that constitute our attack vectors, we summarise common properties of modern architectures. The vast majority of general purpose systems use set-associative caches:

- (i) Data is transferred between memory and cache in blocks of fixed size, called cache lines.
- (ii) The memory addresses are logically partitioned into sets of lines that are congruent wrt. a set index; usually set index depends on either virtual addresses (then the cache is called virtually indexed) or physical addresses (then the cache is called physically indexed);
- (iii) The cache contains a number of ways which can hold one corresponding line for every set index.
- (iv) A cache line stores both the data, the corresponding physical memory location (the tag) and a dirty flag (which indicates if the line has been changed since it was read from memory).

Caches are used by processors to store frequently accessed information and thus to reduce the number of accesses to main memory. A processor can use separate instruction and data caches in a Harvard arrangement (e.g. the L1 cache in ARM Cortex A7) or unified caches (e.g. the L2 cache in ARM Cortex A7). Not all memory areas should be cached; for instance, accesses to addresses representing registers of memory mapped devices should always be directly sent to the main

```

A1) write(VA_c, 1)
A2) write(VA_nc, 0)
A3) call victim
A4) D = read(VA_nc)

V1) if secret
    access(VA3)
else
    access(VA4)

V2) access(VA3+secret)

A1) invalidate(VA_c)
A2) write(VA_nc, 0)
A3) D = read(VA_c)
A4) write(VA_nc, 1)
A5) call victim
A6) D = read(VA_c)

```

(a) For write-back caches with inertia and lazy write (b) For write-through caches or caches that do not guarantee inertia or lazy write

Figure C.1: Confidentiality threat due to data-cache

memory subsystem. For this reason, modern Memory Management Units (MMUs) allow to configure, via the page tables, the caching policy on a per-page basis, allowing a fine-grained control over if and how areas of memory are cached.

In Sections C.3.1, C.3.2 and C.3.3 we present three new attack vectors that depends on misconfigurations of systems and caches. These attacks exploit the following behaviours:

- Mismatched cacheability attributes; if the data cache reports a hit on a memory location that is marked as non-cacheable, the cache might access the memory disregarding such hit. ARM calls this event “unexpected cache hit”.
- Self-modifying code; even if the executable code is updated, the processor might execute the old version of it if this has been stored in the instruction cache.

The attacks can be used to threaten both confidentiality and integrity of a target system. Moreover, two of them use new storage channels suitable to mount access driven attacks. This is particularly concerning, since so far only a noisy timing channel could be used to launch attacks of this kind, which makes real implementations difficult and slow. The security threats are particularly severe whenever the attacker is able to (directly or indirectly) produce the misconfigurations that enable the new attack vectors, as described in Section C.3.4.

C.3.1 Attacking Confidentiality Using Data-Caches

Here we show how an attacker can use mismatched cacheability attributes to mount access-driven cache attacks; i.e. measuring which data-cache lines are evicted by the execution of the victim.

We use the program in Figure C.1a to demonstrate the attacker programming model. For simplicity, we assume that the cache is physically indexed, it has only one way and that it uses the write allocate/write back policy. We also assume that the attacker can access the virtual addresses va_c and va_{nc} , both pointing to the physical address pa ; va_c is cacheable while va_{nc} is not. The attacker writes 1 and 0 into the virtual addresses va_c and va_{nc} respectively, then it invokes the victim. After the victim returns, the attacker reads back from the address va_{nc} .

Let idx be the line index corresponding to the address pa . Since va_c is cacheable, the instruction in $A1$ stores the value 1 in the cache line indexed by idx , the line is flagged as dirty and its tag is set to pa . When the instruction in $A2$ is executed, since va_{nc} is non-cacheable, the system ignores the “unexpected cache hit” and the value 0 is directly written into the memory, bypassing the cache. Now, the value stored in main memory after the execution of the victim depends on the behaviour of the victim itself; if the victim accesses at least one address whose line index is idx , then the dirty line is evicted and the value 1 is written back to the memory; otherwise the line is not evicted and the physical memory still contains the value 0 in pa . Since the address is non-cacheable, the value that is read from va_{nc} in $A4$ depends on the victim’s behaviour.

This mechanism enables the attacker to probe if the line index idx is evicted by the the victim. If the attacker has available a pair of aliases (cacheable and non-cacheable) for every cache line index, the attacker is able to measure the list of cache lines that are accessed by the victim, thus it can mount an access-driven cache attack. The programs V1 and V2 in Figure C.1a exemplify two victims of such attack; in both cases the lines evicted by the programs depend on a confidential variable $secret$ and the access-driven cache attack can extract some bits of the secret variable.

Note that we assumed that the data cache (i) is “write-back”, (ii) has “inertia” and (iii) uses “lazy write”. That is, (i) writing is done in the cache and the write access to the memory is postponed, (ii) the cache evicts a line only when the corresponding space is needed to store new content, and (iii) a dirty cache line is written back only when it is evicted. This is not necessarily true; the cache can be write-through or it can (speculatively) write back and clean dirty lines when the memory bus is unused. Figure C.1b presents an alternative attack whose success does not depend on this assumption. The attacker ($A1-A3$) stores the value 0 in the cache, by invalidating the corresponding line, writing 0 into the memory and reading back the value using the cacheable virtual address. Notice that after step $A3$ the cache line is clean, since the attacker used the non-cacheable virtual alias to write the value 0. Then, the attacker writes 1 into the memory, bypassing the cache. The value read in $A6$ using the cacheable address depends on the behaviour of the victim; if the victim accesses at least one address whose line index is idx , then the cache line for pa is evicted and the instruction in $A6$ fetches the value from the memory, yielding the value 1; otherwise the line is not evicted and the cache still contains the value 0 for pa .

```

V1) D = access(VA_c)
A1) write(VA_nc, 1)
V2) D = access(VA_c)
V3) if not policy(D)
      reject
      [evict VA_c]
V4) use(VA_c)

```

Figure C.2: Integrity threat due to data-cache

C.3.2 Attacking Integrity Using Data-Caches

Mismatched cacheability attributes may also produce integrity threats, by enabling an attacker to modify critical data in an unauthorized or undetected manner. Figure D.1a demonstrates an integrity attack. Again, we assume that the data-cache is direct-mapped, that it is physically indexed and that its write policy is write allocate/write back. For simplicity, we limit the discussion to the L1 caches. In our example, va_c and va_{nc} are virtual addresses pointing to the same memory location pa ; va_c is the cacheable alias while va_{nc} is non-cacheable. Initially, the memory location pa contains the value 0 and the corresponding cache line is either invalid or the line has valid data but it is clean. In a sequential model where reads and writes are guaranteed to take place in program order and their effects are instantly visible to all system components, the program of Figure D.1a has the following effects: V1) a victim accesses address va_c , reading 0; A1) the attacker writes 1 into pa using the virtual alias va_{nc} ; V2) the victim accesses again va_c , this time reading 1; V3) if 1 does not respect a security policy, then the victim rejects it; otherwise V4) the victim uses 1 as the input for a security-relevant functionality.

On a real processor with a relaxed memory model the same system can behave differently, in particular: V1) using va_c , the victim reads initial value 0 from the memory at the location pa and fills the corresponding line in the cache; A1) the attacker use va_{nc} to write 1 directly into the memory, bypassing the cache; V2) the victim accesses again va_c , reading 0 from the cache; V3) the policy is evaluated based on 0; possibly, the cache line is evicted and, since it is not dirty, the memory is not affected; V4) the next time that the victim accesses pa it will read 1 and will use this value as input of the functionality, but 1 has not been checked against the policy. This enables an attacker to bypass a reference monitor, here represented by the check of the security policy, and to inject unchecked input as parameter of security-critical functions.

C.3.3 Attacking Confidentiality Using Instruction Caches

Similar to data caches, instruction caches can be used to mount access-driven cache attacks; in this case the attacker probes the instruction cache to extract information about the victim execution path.

Our attack vector uses self-modifying code. The program in Figure C.3 demon-

```

A1) jmp A8
A2) write(&A8, {R0=1})
A3) call victim
A4) jmp A8
A5) D = R0
...
A8) R0=0
A9) return

V1) if secret
      jmp f1
    else
      jmp f2

```

Figure C.3: Confidentiality threat due to instruction-cache

strates the principles of the attack. We assume that the instruction cache is physically indexed and that it has only one way. We also assume that the attacker's executable address space is cacheable and that the processor uses separate instruction and data caches.

Initially, the attacker's program contains a function at the address $A8$ that writes 0 into the register $R0$ and immediately returns. The attacker starts in $A1$, by invoking the function at $A8$. Let idx be the line index corresponding to the address of $A8$: Since the executable address space is cacheable, the execution of the function has the side effect of temporarily storing the instructions of the function into the instruction cache. Then ($A2$), the attacker modifies its own code, overwriting the instruction at $A8$ with an instruction that updates register $R0$ with the value 1. Since the processor uses separate instruction and data caches the new instruction is not written into the instruction cache. After that the victim completes the execution of its own code, the attacker ($A4$) re-executes the function at $A8$. The instruction executed by the second invocation of the function depends on the behaviour of the victim: if the execution path of the victim contains at least one address whose line index is idx then the attacker code is evicted, the second execution of the function fetches the new instruction from the memory and the register is updated with the value 1; otherwise, the attacker code is not evicted and the second execution of the function uses the old code updating the register with 0.

In practice, the attacker can probe if the line index idx is evicted by the victim. By repeating the probing phase for every cache line, the attacker can mount access-driven instruction cache attacks. The program $V1$ in Figure C.3 exemplifies a victim of such attack, where the control flow of the victim (and thus the lines evicted by the program) depends on a confidential variable $secret$.

C.3.4 Scenarios

In this section we investigate practical applications and limits of the attack vectors. To simplify the presentation, we assumed one-way physically indexed caches. However, all attacks above can be straightforwardly applied to virtually indexed caches. Also, the examples can be extended to support multi-way caches if the way-allocation strategy of the cache does not depend on the addresses that are accessed: the attacker repeats the cache filling phase using several addresses that are all mapped to the same line index. The attack presented in Section C.3.3 also assumes that the processor uses separate instruction and data caches. This is the case in most modern processors, since they usually use the “modified Harvard architecture”. Modern x64 processors, however, implement a snooping mechanism that invalidates corresponding instruction cache lines automatically in case of self-modifying code ([119], Vol. 3, Sect. 11.6); in such a scenario the attack cannot succeed.

The critical assumptions of the attacks are the ability of building virtual aliases with mismatched cacheability attributes (for the attacks in Sections C.3.1 and C.3.2) and the ability of self-modifying code (for the attack in Section C.3.3). These assumptions can be easily met if the attacker is a (possibly compromised) operating system and the victim is a colocated guest in a virtualized environment. In this case, the attacker is usually free to create several virtual aliases and to self-modify its own code. A similar scenario consists of systems that use specialised hardware to isolate security-critical components (like SGX and TrustZone), where a malicious operating system shares the caches with trusted components. Notice also that in case of TrustZone and hardware assisted virtualization, the security software (e.g. the hypervisor) is not informed about the creation of setups that enable the attack vectors, since it usually does not interfere with the manipulation of the guest page tables.

In some cases it is possible to enable the attack vectors even if the attacker is executed in non-privileged mode. Some operating systems can allow user processes to reconfigure cacheability of their own virtual memory. The main reason of this functionality is to speed up some specialised computations that need to avoid polluting the cache with data that is accessed infrequently [171]. In this case two malicious programs can collude to build the aliases having mismatched attributes.

Since buffer overflows can be used to inject malicious code, modern operating systems enforce the executable space protection policy: a memory page can be either writable or executable, but it can not be both at the same time. However, to support just in time compilers, the operating systems allow user processes to change at run-time the permission of virtual memory pages, allowing to switch a writable page into an executable and vice versa (e.g. Linux provides the syscall “mprotect”, which changes protection for a memory page of the calling process). Thus, the attack of Section C.3.3 can still succeed if: (i) initially the page containing the function *A8* is executable, (ii) the malicious process requests the operating system to switch the page as writable (i.e. between step *A1* and *A2*) and (iii) the process

requests the operating system to change back the page as executable before re-executing the function (i.e. between step *A2* and *A4*). If the operating system does not invalidate the instruction cache whenever the permissions of memory pages are changed, the confidentiality threat can easily be exploited by a malicious process.

In Sections C.2 and D.2 we provide a summary of existing literature on side channel attacks that use caches. In general, every attack (e.g. [6, 232, 158]) that is access-driven and that has been implemented by probing access times can be reimplemented using the new vectors. However, we stress that the new vectors have two distinguishing characteristics with respect to the time based ones: (i) the probing phase does not need the support of an external measurement, (ii) the vectors build a cache-based storage channel that has relatively low noise compared channels based on execution time which depend on many other factors than cache misses, e.g., TLB misses and branch mispredictions.

In fact, probing the cache state by measuring execution time requires the attacker to access the system time. If this resource is not directly accessible in the execution level of the attacker, the attacker needs to invoke a privileged function that can introduce delays and noise in the cache state (e.g. by causing the eviction from the data cache when accessing internal data-structures). For this reason, the authors of [232] disabled the timing virtualization of XEN (thus giving the attacker direct access to the system timer) to demonstrate a side channel attack. Finally, one of the storage channels presented here poses integrity threats clearly outside the scope of timing based attacks.

C.4 Case Studies

To substantiate the importance of the new attack vectors, and the need to augment the verification methodology to properly take caches and cache attributes into account, we examine the attack vectors in practice. Three cases are presented: A malicious OS that extracts a secret AES key from a cryptoservice hosted in TrustZone, a malicious paravirtualized OS that subverts the memory protection of a hypervisor, and a user process that extracts the exponent of a modular exponentiation procedure executed by another process.

C.4.1 Extraction of AES Keys

AES [70] is a widely used symmetric encryption scheme that uses a succession of rounds, where four operations (SubBytes, ShiftRows, MixColumn and AddRound-Key) are iteratively applied to temporary results. For every round *i*, the algorithm derives the sub key K_i from an initial key k . For AES-128 it is possible to derive k from any sub key K_i .

Traditionally, efficient AES software takes advantage of precomputed SBox tables to reach a high performance and compensate the lack of native support to low-level finite field operations. The fact that disclosing access patterns to the SBoxes

can make AES software insecure is well known in literature (e.g. [224, 208, 6]). The existing implementations of these attacks probe the data cache using time channels, here we demonstrate that such attacks can be replicated using the storage channel described in Section C.3.1. With this aim, we implement the attack described in [158].

The attack exploits a common implementation pattern. The last round of AES is slightly different from the others since the MixColumn operation is skipped. For this reason, implementations often use four SBox tables T_0, T_1, T_2, T_3 of 1KB for all the rounds except the last one, whereas a dedicated T_4 is used. Let c be the resulting cipher-text, n be the total number of rounds and x_i be the intermediate output of the round i . The last AES round computes the cipher-text as follows:

$$c = K_n \oplus ShiftRows(SubBytes(x_{n-1}))$$

Instead of computing $ShiftRows(SubBytes(x_{n-1}))$, the implementation accesses the precomputed table T_4 according to an index that depends on x_{n-1} . Let $b[j]$ denote the j -th byte of b and $[T_4 \text{ output}]$ be one of the actual accesses to T_4 , then

$$c[j] = K_n[j] \oplus [T_4 \text{ output}] .$$

Therefore, it is straightforward to compute K_n knowing the cipher-text and the entry yielded by the access to T_4 :

$$K_n[j] = c[j] \oplus [T_4 \text{ output}]$$

Thus the challenge is to identify the exact $[T_4 \text{ output}]$ for a given byte j . We use the “non-elimination” method described in [158]. Let L be a log of encryptions, consisting of a set of pairs (c_l, e_l) . Here, c_l is the resulting cipher-text and e_l is the set of cache lines accessed by the AES implementation. We define $L_{j,v}$ to be the subset of L such that the byte j of the cipher-text is v :

$$L_{j,v} = \{(c_l, e_l) \in L \text{ such that } c_l[j] = v\}$$

Since $c[j] = K_n[j] \oplus [T_4 \text{ output}]$ and the key is constant, if the j -th byte of two cipher-texts have the same value then the accesses to T_4 for such cipher-text must contain at least one common entry. Namely, the cache line accessed by the implementation while computing $c[j] = K_n[j] \oplus [T_4 \text{ output}]$ is (together with some false positives) in the non-empty set

$$E_{j,v} = \bigcap_{(c_l, e_l) \in L_{j,v}} e_l$$

Let $T_4^{j,v}$ be the set of distinct bytes of T_4 that can be allocated in the cache lines $E_{j,v}$. Let v, v' be two different values recorded in the log for the byte j . We know that exist $t_4^{i,v} \in T_4^{j,v}$ and $t_4^{i,v'} \in T_4^{j,v'}$ such that $v = K_n[j] \oplus t_4^{i,v}$ and $v' = K_n[j] \oplus t_4^{i,v'}$. Thus

$$v \oplus v' = t_4^{j,v} \oplus t_4^{j,v'}$$

This is used to iteratively shrink the sets $T_4^{j,v}$ and $T_4^{j,v'}$ by removing the pairs that do not satisfy the equation. The attacker repeats this process until for a byte value v the set $T_4^{j,v}$ contains a single value; then the byte j of key is recovered using $K_n[j] = v \oplus t_4^{j,v}$. Notice that the complete process can be repeated for every byte without gathering further logs and that the attacker does not need to know the plain-texts used to produce the cipher-texts.

We implemented the attack on a Raspberry Pi 2 [179], because this platform is equipped with a widely used CPU (ARM Cortex A7) and allows to use the TrustZone extensions. The system starts in TrustZone and executes the bootloader of our minimal TrustZone operating system. This installs a secure service that allows an untrusted kernel to encrypt blocks (e.g. to deliver packets over a VPN) using a secret key. This key is intended to be confidential and should not be leaked to the untrusted software. The trusted service is implemented using an existing AES library for embedded devices [227], that is relatively easy to deploy in the resource constrained environment of TrustZone. However, several other implementations (including OpenSSL [161]) expose the same weakness due to the use of precomputed SBoxes. The boot code terminates by exiting TrustZone and activating the untrusted kernel. This operating system is not able to directly access the TrustZone memory but can invoke the secure service by executing Secure Monitor Calls (SMC).

In this setting, the attacker (the untrusted kernel), which is executed as privileged software outside TrustZone, is free to manipulate its own page tables (which are different from the ones used by the TrustZone service). Moreover, the attacker can invalidate and clean cache lines, but may not use debugging instructions to directly inspect the state of the caches.

The attacker uses the algorithm presented in Figure C.1b, however several considerations must be taken into account to make the attack practical. The attacker repeats the filling and probing phases for each possible line index (128) and way (4) of the data-cache. In practice, since the cache eviction strategy is pseudo random, the filling phase is also repeated several times, until the L1 cache is completely filled with the probing data (i.e. for every pair of virtual addresses used, accessing to the two addresses yield different values).

On Raspberry Pi 2, the presence of a unified L2 cache can obstruct the probing phase: even if a cache line is evicted from the L1 cache by the victim, the system can temporarily store the line into the L2 cache, thus making the probing phase yield false negatives. It is in general possible to extend the attack to deal with L2 caches (by repeating the filling and probing phases for every line index and way of the L2 cache subsystem), however, in Raspberry Pi 2 the L2 cache is shared between the CPU and the GPU, introducing a considerable amount of noise in the measurements. For this reason we always flushes the L2 cache between the step $A5$ and $A6$ of the attack. We stress that this operation can be done by the privileged software outside TrustZone without requiring any support by TrustZone itself.

To make the demonstrator realistic, we allow the TrustZone service to cache its

own stack, heap, and static data. This pollutes the data extracted by the probing phase of the attack: it can now yield false positives due to access of the victim to such memory areas. The key extraction algorithm can handle such false positives, but we decide to filter them out to speed up the analysis phase. For this reason, the attacker first identifies the cache lines that are frequently evicted independently of the resulting cipher-text (e.g. lines where the victim stack is probably allocated) and removes them from the sets $E_{j,v}$. As common, the AES implementation defines the SBox tables as consecutive arrays. Since they all consists of 1 KB of data, the cache lines where different SBoxes are allocated are non-overlapping, helping the attacker in the task of reducing the sets $E_{j,v}$ to contain a single line belonging to the table T_4 and of filtering out all evictions that are due the previous rounds of AES.

For practical reasons we implemented the filling and probing phase online, while we implemented the key extraction algorithm as a offline Python program that analyses the logs saved by the online phase. The complete online phase (including the set-up of the page tables) consists of 552 lines of C, while the Python programs consists of 152 lines of code. The online attacker generates a stream of random 128 bits plain-texts and requests to the TrustZone service their encryption. Thus, the frequency of the attacker’s measurements isolates one AES encryption of one block per measurement. Moreover, even if the attacker knows the input plain-texts, they are not used in the offline phase. We repeated the attack for several randomly generated keys and in the worst case, the offline phase recovered the complete 128-bit key after 850 encryption in less than one second.

C.4.2 Violating Spatial Isolation in a Hypervisor

A hypervisor is a low-level execution platform controlling accesses to system resources and is used to provide isolated partitions on a shared hardware. The partitions are used to execute software with unknown degree of trustworthiness. Each partition has access to its own resources and cannot encroach on protected parts of the system, like the memory used by the hypervisor or the other partitions. Here we demonstrate that a malicious operating system (guest) running on a hypervisor can gain illicit access to protected resources using the mechanism described in Section C.3.2.

As basis for our study we use a hypervisor [157] that has been formally verified previously with respect to a cache-less model. The hypervisor runs on an ARMv7 Cortex-A8 processor [63], where both L1 and L2 caches are enabled. On ARMv7 the address translation depends on the page tables stored in the memory. Entries of the page tables encode a virtual-to-physical mapping for a memory page as well as access permissions and cacheability setting. On Cortex-A8 the MMU consults the data cache before accessing the main memory whenever a page table descriptor must be fetched.

The architecture is paravirtualized by the hypervisor for several guests. Only the hypervisor is executing in privileged mode, while the guests are executed in non-

privileged mode and need to invoke hypervisor functionality to alter the critical resources of the system, like page tables.

A peculiarity of the hypervisor (and others [26]) that makes it particularly relevant for our purpose is the use of so-called direct paging [157]. Direct paging enables a guest to manage its own memory space with assistance of the hypervisor. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Then, the guest uses dedicated hypervisor calls to effectuate and monitor the transition of page tables between passive and active state. The hypervisor provides a number of system calls that support the allocation, deallocation, linking, and activation of guest page tables. These calls need to read the content of page tables that are located in guest memory and ensure that the proposed MMU setup does not introduce any illicit access grant. Thus the hypervisor acts as a reference monitor of the page tables.

As described in Section C.3.2, on a Cortex-A8 processor sequential consistency is not guaranteed if the same memory location is accessed by virtual aliases with mismatched cacheability attributes. This opens up for vulnerabilities. The hypervisor may check a page table by fetching its content from the cache. However, if the content of the page table in the cache is clean and different from what has been placed by the attacker in the main memory and the page table is later evicted from the cache, the MMU will use a configuration that is different from what has been validated by the hypervisor.

Figure C.4 illustrates how a guest can use the aliasing of the physical memory to bypass the validation needed to create a new page table. Hereafter we assume that the guest and the hypervisor use two different virtual addresses to point to the same memory location. Initially, the hypervisor (1) is induced to load a valid page table in the cache. This can be done by writing a valid page table, requesting the hypervisor to verify and allocate it and then requesting the hypervisor to deallocate the table. Then, the guest (2) stores an invalid page table in the same memory location. If the guest uses a non-cacheable virtual alias, the guest write (3) is directly applied to the memory bypassing the cache. The guest (4) requests the hypervisor to validate and allocate this memory area, so that it can later be used as page table for the MMU. At this point, the hypervisor is in charge of verifying that the memory area contains a valid page table and of revoking any direct access of the guest to this memory. In this way, a validated page table can be later used securely by the MMU. Since the hypervisor (4) accesses the same physical location through the cache, it can potentially validate stale data, for example the ones fetched during the step (1). At a later point in time, the validated data is evicted from the cache. This data is not written back to the memory since the hypervisor has only checked the page table content and thus the corresponding cache lines are clean. Finally, the MMU (5) uses the invalid page table and its settings become untrusted.

Note that this attack is different from existing “double mapping” attacks. In double-mapping attacks the same physical memory is mapped “simultaneously” to multiple virtual memory addresses used by different agents; the attack occurs when

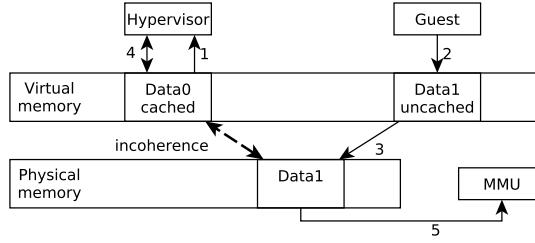


Figure C.4: Compromising integrity of a direct paging mechanism using incoherent memory. The MMU is configured to use a page table that was not validated by the hypervisor.

the untrusted agent owns the writable alias, thus being able to directly modify the memory accessed by the trusted one. Here, the attacker exploits the fact that the same physical memory is first allocated to the untrusted agent and then re-allocated to the trusted one. After that the ownership is transferred (after step A1), the untrusted agent has no mapping to this memory area. However, if the cache contains stale data the trusted agent may be compromised. Moreover, the attack does not depend on misconfiguration of the TLBs; the hypervisor is programmed to completely clean the TLBs whenever the MMU is reconfigured.

We implemented a malicious guest that managed to bypass the hypervisor validation using the above mechanism. The untrusted data, that is used as configuration of the MMU, is used to obtain writable access to the master page table of the hypervisor. This enables the attacker to reconfigure its own access rights to all memory pages and thus to completely take over the system.

Not all hypervisors are subject to this kind of vulnerability. For example, if a hypervisor uses shadow paging, then guest pages are copied into the hypervisor's own memory where they are transformed into so-called shadow page tables. The guest has no access to this memory area and the hypervisor always copies cached data (if present), so the attack described above cannot be replicated. On the other hand, the adversary can still attack secure services hosted by the hypervisor, for example a virtual machine introspector. In [51] the hypervisor is used to implement a run-time monitor to protect an untrusted guest from its internal threats. The monitor is deployed in a separate partition to isolate it from the untrusted guest. The policy enforced by the monitor is executable space protection: each page in the memory can be either writable or executable but not both at the same time. The monitor, via the hypervisor, intercepts all changes to the executable codes. This allows to use standard signature checking to prevent code injection. Each time the guest operating system tries to execute an application, the monitor checks if the binary of the application has a valid signature. In case the signature is valid, the

```

y := 1
for i = m down to 1
    y = Square(y)
    y = ModReduce(y, N)
    if e_i == 1
        y = Mult(y, x)
    y = ModReduce(y, N)

```

Figure C.5: Square and multiply algorithm

monitor requests the hypervisor to make executable the physical pages that contain the binary code. The security of this system depends on the fact that the adversary cannot directly modify a validated executable due to executable space protection. However, if a memory block of the application code is accessed using virtual aliases with mismatched cacheability attributes, the untrusted guest can easily mislead the monitor to validate wrong data and execute unsigned code.

C.4.3 Extraction of Exponent From a Modular Exponentiation Procedure

The square and multiply algorithm of Figure C.5 is often used to compute the modular exponentiation $x^e \bmod N$, where $e_m \dots e_1$ are the bits of the binary representation of e . This algorithm has been exploited in access-driven attacks, since the sequence of function calls directly leaks e , which corresponds to the private key in several decryption algorithms. Here we demonstrate that an attacker that is interleaved with a victim can infer e using the storage channel described in Section C.3.3.

The attack was implemented on Raspberry Pi 2. We build a setting where a malicious process (e.g. a just in time compiler) can self-modify its own code. Moreover, we implement a scheduler that allows the attacker to be scheduled after every loop of the victim.¹

The attacker uses the vector presented in Figure C.3, repeating the filling and probing phases for every way of the instruction cache and for every line index where the code of the functions `Mult` and `ModReduce` can be mapped. Due to the separate instruction and data L1 caches, the presence of the L2 cache does not interfere with the probing phase. However, we must ensure that the instruction overwritten in the step (A2) does not sit in the L1 data-cache when the step (A4) is executed. Since user processes cannot directly invalidate or clean cache lines, we satisfy this requirement by adding a further step (A3.b). This step writes several addresses whose line indices in the L1 data-cache are the same of the address &A8, thus forcing the eviction from the L1 data-cache of the line that has contains the instruction stored at &A8.

¹Forcing the scheduler of a general purpose OS to grant such high frequency of measurements is out of the scope of this paper. The interested reader can refer to [158, 232].

We repeated the attack for several randomly generated values of e and in each case the attacker correctly identified the execution path of the victim. This accuracy is due to the simple environment (no other process is scheduled except the victim and the attacker) and the lack of noise that is typical in attacks that use time channels.

C.5 Countermeasures

Literature on access-based timing channel attacks suggests a number of well-known countermeasures. Specifically, for attacks on the confidentiality of AES encryption, a rather comprehensive list of protective means is provided in [224]. Some of the approaches are specific to AES, e.g., using registers instead of memory or dedicated hardware instructions for the SBox table look-up. Others are specific to the timing attack vector, e.g., reducing the accuracy of timing information available to the attacker. Still, there are well-known solutions addressing the presence of caches in general, thus they are suitable to defend against attacks built on the cache storage channel described in this paper.

In what follows we identify such known general countermeasures (Sections C.5.1 and C.5.3.1-5) and propose new ones that are specific to the attack vector using uncacheable aliases (Sections C.5.2, C.5.3.6, and C.5.4). In addition it is examined which countermeasures are suitable to protect against the integrity threat posed by incoherent aliases in the memory system and propose a fix for the hypervisor example.

Different countermeasures are evaluated by implementing them for the AES and hypervisor scenarios introduced in the previous section and analysing their performance. The corresponding benchmark results are shown in Tables C.1 and C.2. Since our main focus is on verifying systems in the presence of caches, for each group of countermeasures we also sketch how a correctness proof would be conducted. Naturally, such proofs require a suitable model of the memory system including instruction and data caches.

It should be emphasised that the verification of the countermeasures is meant to be performed separately from the verification of the overall system which is usually assuming a much simpler memory model for feasibility. The goal is to show that the countermeasures neutralise the cache storage channels and re-establish a coherent memory model. The necessary program verification conditions from such a proof can then be incorporated into the overall verification methodology, supporting its soundness.

C.5.1 Disabling Cacheability

The simplest way to eliminate the cache side channel is to block an attacker from using the caches altogether. In a virtualization platform, like an operating system or a hypervisor, this can be achieved by enforcing the memory allocated to untrusted guest partitions to be uncacheable. Consequently, cache-driven attacks on

confidentiality and integrity of a system are no longer possible. Unfortunately, this countermeasure comes at great performance costs, potentially slowing down a system by several orders of magnitude. On the other hand, a proof of the correctness of the approach is straight-forward. Since the attacker cannot access the caches, they are effectively invisible to him. The threat model can then be specified using a coherent memory semantics that is a sound abstraction of a system model where caches are only used by trusted code.

C.5.2 Enforcing Memory Coherency

Given the dramatic slowdown expected for a virtualization platform, it seems out of the question to completely deny the use of caches to untrusted guests. Nevertheless, the idea of enforcing that guest processes cannot break memory coherency through uncacheable aliases still seems appealing.

Always Cacheable Guest Memory

When making all guest memory uncacheable is prohibitively expensive, an intuitive alternative could be to just make all guest memory cacheable. Indeed, if guests are user processes in an operating system this can be easily implemented by adapting the page table setup for user processes accordingly, i.e., enforcing cacheability for all user pages. Then user processes cannot create uncacheable aliases to measure cache contents and start cache-based time-of-check-to-time-of-use attacks on their host operating system.

However, for hypervisors, where guests are whole operating systems, the approach has several drawbacks. First of all, operating systems are usually controlling memory mapped I/O devices which should be operated through uncacheable memory accesses. If a hypervisor would make all memory accesses of a guest OS cacheable, the OS will not be able to properly control I/O devices and probably not work correctly. Thus, making all untrusted guest memory cacheable only works for (rather useless) operating systems that do not control I/O devices. Furthermore, there are cases when a guest can optimise its performance by making seldomly used pages uncacheable [171].

$C \oplus U$ Policy

Instead of making all guest pages cacheable, a hypervisor could make sure that at all times a given physical page can either be accessed in cacheable or uncacheable mode ($C \oplus U$ policy). To this end it would need to monitor the page table setup of the guests and forbid them to define both cacheable and uncacheable aliases of the same physical address. Then guests may set up uncacheable virtual pages only if no cacheable alias exists for the targeted physical page. Moreover, the hypervisor has to flush a cacheable page from the caches when it becomes uncacheable, in order to remove stale copies of the page that might be abused to set up an alias-driven

LMbench micro benchmark	Native	Hyp	ACPT	SelFl	Flush
null syscall	0.41	1.75	1.76	1.77	1.76
read	0.84	2.19	2.20	2.20	2.38
write	0.74	2.09	2.10	2.15	2.22
stat	3.22	5.61	5.50	5.89	5.92
fstat	1.19	2.53	2.55	2.56	2.65
open/close	6.73	14.50	14.42	14.86	14.71
select(10)	1.86	3.29	3.30	3.33	3.42
sig handler install	0.85	2.87	2.89	2.92	2.95
sig handler overhead	4.43	14.45	14.48	15.11	14.91
protection fault	2.66	3.73	3.83	3.91	3.70
pipe	21.83	48.78	47.79	47.62	692.91
fork+exit	1978	5106	5126	6148	38787
fork+execve	2068	5249	5248	6285	39029
pagefaults	3.76	11.21	11.12	21.55	332.82
Application benchmark	Native	Hyp	ACPT	SelFl	Flush
tar (500K)	70	70	70	70	190
tar (1M)	120	120	120	120	250
tar (2M)	230	210	200	210	370
dd (10M)	90	140	140	160	990
dd (20M)	190	260	260	570	1960
dd (40M)	330	500	450	600	3830
jpg2gif(5KB)	60	60	60	60	130
jpg2gif(250KB)	920	810	820	830	1230
jpg2gif(750KB)	930	870	870	880	1270
jpg2bmp(5KB)	40	40	40	40	110
jpg2bmp(250KB)	1350	1340	1340	1350	1720
jpg2bmp(750KB)	1440	1420	1420	1430	1790
jpegtrans(270', 5KB)	10	10	10	10	80
jpegtrans(270', 250KB)	220	240	240	250	880
jpegtrans(270', 750KB)	380	400	400	420	1050
bmp2tiff(90 KB)	10	10	10	10	60
bmp2tiff(800 KB)	20	20	20	20	80
ppm2tiff(100 KB)	10	10	10	10	70
ppm2tiff(250 KB)	10	10	10	20	80
ppm2tiff(1.3 MB)	20	30	30	30	90
tif2rgb(200 KB)	10	20	20	20	120
tif2rgb(800 KB)	40	40	40	50	270
tif2rgb(1.200 MB)	130	160	160	180	730
sox(aif2wav 100KB)	20	20	20	30	140
sox(aif2wav 500KB)	40	60	60	60	180
sox(aif2wav 800KB)	60	100	100	110	220

Table C.1: Hypervisor Micro and Application Benchmarks. LMbench micro benchmarks [μs] and application benchmarks [ms] for the Linux kernel v2.6.34 running natively on BeagleBone Black, paravirtualized on the hypervisor without protection against the integrity threat (Hyp), with always cacheable page tables (ACPT), with selective flushing (SelFl), and with full cache flushes on entry (Flush).

cache attack. In this way, the hypervisor would enforce memory coherency for the guest memory by making sure that no content from uncacheable guest pages is ever cached and for cacheable pages cache entries may only differ from main memory if they are dirty.

A Trust-zone cryptoservice that intends to prevent a malicious OS to use memory incoherency to measure the Trust-zone accesses to the cache can use TZ-RKP [18] and extend its run-time checks to force the OS to respect the $C \oplus U$

policy.

Second-Stage MMU

Still, for both the static and the dynamic case, the $C \oplus U$ policy may be expensive to implement for fully virtualizing hypervisors that rely on a second stage of address translation. For example, the ARMv8 architecture provides a second stage MMU that is controlled by the hypervisor, while the first stage MMU is controlled by the guests. Intermediate physical addresses provided by the guests are then remapped through the second stage to the actual physical address space. The mechanism allows also to control the cacheability of the intermediate addresses, but it can only enforce non-cacheability. In order to enforce cacheability, the hypervisor would need to enforce it on the first stage of translation by intercepting the page table setup of its guests, which creates an undesirable performance overhead and undermines the idea of having two independently operated stages of address translation.

$W \oplus X$ Policy

Unfortunately, enforcing cacheability of memory accesses does not protect against the instruction-cache-based confidentiality threat described earlier. In order to prevent an attacker from storing incoherent copies for the same instruction address in the memory system, the hypervisor would also need to prohibit self-modifying code for the guests, i.e., ensure that all guest pages are either writable or executable ($W \oplus X$ policy). Since operating systems regularly use self-modification, e.g., when installing kernel updates or swapping in pages, the association of pages to the executable or writable attribute is dynamic as well and must be monitored by the hypervisor. It also needs to flush instruction caches when an executable page becomes writable.

Overall, the solutions presented above seem to be more suitable for paravirtualizing hypervisors, that are invoked by the guests explicitly to configure their virtual memory. Adding the required changes to the corresponding MMU virtualization functionality seems straightforward. In fact, for the paravirtualizing hypervisor presented in this paper a tamper-proof security monitor has been implemented and formally verified, which enforces executable space protection on guest memory and checks code signatures in order to protect the guests from malicious code injection [51].

Always Cacheable Page Tables

To protect the hypervisor against the integrity threat a lightweight specialization of the $C \oplus U$ policy introduced above was implemented. It is based on the observation that uncacheable aliases can only subvert the integrity of the hypervisor if they are constructed for the inputs of its MMU virtualization functions. Thus the hypervisor needs only to enforce the $C \oplus U$ policy, and consequently memory coherency, on its inputs. While this can be achieved by flushing the caches appropriately (see

AES encryption	5 000 000 × 16B		10 000 × 8KB	
	Time	Throughput	Time	Throughput
Original SBoxes	23s	3.317 MB/s	13s	6.010 MB/s
Compact Last SBox	24s	3.179 MB/s	16s	4.883 MB/s
Scrambled Last SBox	30s	2.543 MB/s	20s	3.901 MB/s
Uncached Last SBox	36s	2.119 MB/s	26s	3.005 MB/s
Scrambled All SBoxes	132s	0.578 MB/s	125s	0.625 MB/s
Uncached All SBoxes	152s	0.502 MB/s	145s	0.539 MB/s

Table C.2: AES Encryption Benchmarks. AES encryption on Raspberry Pi 2 of one block (128 bits = 16 Bytes) and 512 blocks for different SBox layouts.

Section C.5.3), a more efficient approach is to allocate the page tables of the guests in regions that are always cacheable. These regions of physical memory are fixed for each guest and the hypervisor only validates a page table for the guest if it is allocated in this area. In all virtual addresses mapping to the area are forced to be cacheable. Obviously, also the guest system needs to be adapted to support the new requirement on the allocation of page tables. However, given a guest system that was already prepared to run on the original hypervisor, the remaining additional changes should be straight-forward. For instance, the adaptation of the hypervisor required changes to roughly 35 LoC in the paravirtualized Linux kernel and an addition of 45 LoC to the hypervisor for the necessary checks.

The performance of the hypervisor with always cacheable page tables (ACPT) can be observed in Table C.1. Compared to the original hypervisor there are basically no performance penalties. In some cases the new version even outperforms the original hypervisor, due to the ensured cacheability of page tables. It turns out that in the evaluated Linux kernel, page tables are not always allocated in cacheable memory areas. The correctness of the approach is discussed in detail in Section C.6. The main verification condition to be discharged in a formal proof of integrity is that the hypervisor always works on coherent memory, hence any correctness proof based on a coherent model also holds in a more detailed model with caches.

C.5.3 Repelling Alias-Driven Attacks

The countermeasures treated so far were aimed at restricting the behaviour of the attacker to prevent him from harvesting information from the cache channel or break memory coherency in an attack on integrity. A different angle to the problem lies in focusing on the trusted victim process and ways it can protect itself against an unrestricted attacker that is allowed to break memory coherency of its memory and run alias-driven cache attacks. The main idea to protect integrity against such attacks is to (re)establish coherency for all memory touched by the trusted process. For confidentiality, the idea is to adapt the code of the victim in a way that its execution leaks no additional information to the attacker through the cache channel. Interestingly, many of the techniques described below are suitable for both purposes, neutralizing undesirable side effects of using the caches.

Complete Cache Flush

One of the traditional means to tackle cache side channels is to flush all instruction and data caches before executing trusted code. In this way, all aliases in the cache are either written back to memory (in case they are dirty) or simply removed from the cache (in case they are clean). Any kind of priming of the caches by the attacker becomes ineffective since all his cache entries are evicted by the trusted process, foiling any subsequent probing attempts using addresses with mismatched cacheability. Similarly, all input data the victim reads from the attacker's memory are obtained from coherent main memory due to the flush, thus thwarting alias-driven attacks on integrity.

A possible correctness proof that flushing all caches eliminates the information side channel would rely on the assertion that, after the execution of the trusted service, an attacker will always make the same observation using mismatched aliases, i.e., that all incoherent lines were evicted from the cache. Thus he cannot infer any additional knowledge from the cache storage channel. Note, that here it suffices to flush the caches before returning to the attacker, but to protect against the integrity threat, data caches need to be flushed before any input data from the attacker is read.

For performance evaluation the flushing approach was implemented in the AES and hypervisor examples. At each call of an AES encryption or hypervisor function, all data and instruction caches are flushed completely. Naturally this introduces an overhead for the execution of legitimate guest code due to an increased cache miss rate after calls to trusted processes. At the same time the trusted process gets slowed down for the same reason, if normally some of its data and instructions were still allocated in the caches from a previous call. Additionally the flushing itself is often expensive, e.g., for ARM processors the corresponding code has to traverse all cache lines in all ways and levels of cache to flush them individually. That all these overheads can add up to a sizeable delay of even one order of magnitude is clearly demonstrated by the benchmarks given in Tables C.2 and C.1.

Cache Normalization

Instead of flushing, the victim can eliminate the cache information side channel by reading a sequence of memory cells so that the cache is brought into a known state. For instruction caches the same can be achieved by executing a sequence of jumps that are allocated at a set of memory locations mapping to the cache lines to be evicted. In the context of timing channels this process is called normalization. If subsequent memory accesses only hit the normalized cache lines, the attacker cannot observe the memory access pattern of the victim, because the victim always evicts the same lines. However the correctness of this approach strongly depends on the hardware platform used and the replacement policy of its caches. In case several memory accesses map to the same cache line the normalization process may in theory evict lines that were loaded previously. Therefore, in the verification a

detailed cache model is needed to show that all memory accesses of the trusted service hit the cache ways touched during normalization.

Selective Eviction

The normalization method shows that cache side effects can be neutralized without evicting the whole cache. In fact, it is enough to focus on selected cache lines that are critical for integrity or confidentiality. For example, the integrity threat on the hypervisor can be eliminated by evicting the cache lines corresponding to the page table provided by the attacker. The flushing or normalization establishes memory coherency for the hypervisor's inputs, thus making sure it validates the right data. The method of selective flushing was implemented for the hypervisor scenario and benchmark results in Table C.1 show, as one would expect, that it is more efficient than flushing the whole cache, but still slower than our specialized ACPT solution.

To ensure confidentiality in the AES example it suffices to evict the cache lines occupied by the SBoxes. Since the incoherent entries placed in the same cache lines are removed by the victim using flushing or normalization, the attacker subsequently cannot measure key-dependent data accesses to these cache lines. For the modular exponentiation example the same technique can be used, evicting only the lines in the instruction cache where the code of the functions `Mult` and `ModReduce` is mapped.

The correctness of selective eviction of lines for confidentiality depends on the fact that accesses to other lines do not leak secret information through the cache side channel, e.g., for the AES encryption algorithm lines that are not mapped to an SBox are accessed in every computation, independent of the value of the secret key. Clearly, this kind of trace property needs to be added as a verification condition on the code of the trusted service. Then the classic confidentiality property can be established, that observations of the attacker are the same in two computations where only the initial values of the secret are different (non-infiltration [108]).

Secret-Independent Memory Accesses

The last method of eliminating the cache information side channel is a special case of this approach. It aims to transform the victim's code such that it produces a memory access trace that is completely independent of the secret, both for data accesses and instruction fetches. Consequently, there is no need to modify the cache state set up by the attacker, it will be transformed in the same way even for different secret values, given the trusted service receives the same input parameters and all hidden states in the service or the cache model are part of the secret information.

As an example we have implemented a modification of AES suggested in [224], where a 1KB SBox look-up table is scrambled in such a way that a look-up needs to touch all cache lines occupied by the SBox. In our implementation on Raspberry Pi 2 each L1 cache line consists of 64 Bytes, hence a 32bit entry is spread over 16 lines where each line contains two bits of the entry. While the decision which

2 bits from every line are used is depending on the secret AES key, the attacker only observes that the encryption touches the 16 cache lines occupied by the SBox, hence the key is not leaked.

Naturally the look-up becomes more expensive now because a high number of bitfield and shift operations is required to reconstruct the original table entry. For a single look-up, a single memory access is substituted by 16 memory accesses, 32 shifts, 16 additions and 32 bitfield operations. The resulting overhead is roughly 50% if only the last box is scrambled (see Table C.2). This is sufficient if all SBoxes are mapped to the same cache lines and the attacker cannot interrupt the trusted service, probing the intermediate cache state. Scrambling all SBoxes seems prohibitively expensive though, slowing the encryption down by an order of magnitude. However, since the number of operations depends on the number of lines used to store the SBox, if the system has bigger cache lines the countermeasure becomes cheaper.

Reducing the Channel Bandwidth

Finally for the AES example there is a countermeasure that does not completely eliminate the cache side channel, but makes it harder for the attacker to derive the secret key. The idea described in [224] is to use a more compact SBox that can be allocated on less lines, undoing an optimization in wolfSSL for the last round of AES. There the look-up only needs to retrieve one byte instead four, still the implementation word-aligns these bytes to avoid bit masking and shifting. By byte-aligning the entries again, the table shrinks by a factor of four, taking up four lines instead of 16 on Raspberry Pi 2. Since the attacker can distinguish less entries by the cache line they are allocated on, the channel leaks less information. This theory is confirmed in practice where retrieving the AES key required about eight times as many encryptions compared to the original one. At the same time, the added complexity resulted in a performance delay of roughly 23% (see Table C.2).

Detecting Memory Incoherency

A reference monitor (e.g. the hypervisor) can counter the integrity threat by preventing the invocation of the critical functions (e.g. the MMU virtualization functions) if memory incoherency is detected. The monitor can itself use mismatched cache attributes to detect incoherency as follows. For every address that is used as the input of a critical function, the monitor checks if reading the location using the cacheable and non-cacheable aliases yield the same result. If the two reads differs, then memory incoherency is detected and the monitor rejects the request, otherwise the request is processed.

C.5.4 Hardware Based Countermeasures

The cache-based storage channels rely on misbehaviour of the system due to misconfigurations. For this reason, the hardware could directly take care of them. The vector based on mismatched cacheability attributes can be easily made ineffective if the processor does not ignore unexpected cache hits. For example, if a physical address is written using a non-cacheable alias, the processor can invalidate every line having the corresponding tag. Virtually indexed caches are usually equipped with similar mechanisms to guarantee that there can not be aliases inside the cache itself.

Hardware inhibition of the vector that uses the instruction cache can be achieved using a snooping mechanism that invalidates instruction cache lines whenever self-modification is detected, similar to what happens in x64 processors. In architectures that perform weakly ordered memory accesses and aggressive speculative execution, implementing such a mechanism can become quite complex and make the out-of-order execution logic more expensive. There is also a potential slow-down due to misspeculation when instructions are fetched before they are overwritten.

Overall, the presented countermeasures show that a trusted service can be efficiently secured against alias-driven cache attacks if two properties are ensured: (1) for integrity, the trusted service may only access coherent memory (2) for confidentiality, the cache must be transformed in a way such that the attacker cannot observe memory accesses depending on secrets. In next section, a verification methodology presented that aims to prove these properties for the code of the trusted service.

C.6 Verification Methodology

The attacks presented in Section C.4 demonstrate that the presence of caches can make a trustworthy, i.e. formally verified, program vulnerable to both confidentiality and security threats. These vulnerabilities depend on the fact that for some resources (i.e. some physical addresses of the memory) the actual system behaves differently from what is predicted by the formal model: we refer to this misbehaviour as “loss of sequential consistency”.

As basis for the study we assume a low level program (e.g. a hypervisor, a separation kernel, a security monitor, or a TrustZone crypto-service) running on a commodity CPU such as the ARMv7 Cortex A7 of Raspberry Pi 2. We refer to the trusted program as “the kernel”. The kernel shares the system with an untrusted application, henceforth “the application”. We assume that the kernel has been subject to a pervasive formal verification that established its functional correctness and isolation properties using a model that reflects the ARMv7 ISA specification to some level of granularity. For instance for both seL4 and the Prosper kernel the

processor model is based on Anthony Fox’s cacheless L3 model of ARMv7².

We identify two special classes of system resources (read: Memory locations):

- Critical resources: These are the resources whose integrity must be protected, but which the application needs access to for its correct operation.
- Confidential resources: These are the resources that should be read protected against the application.

There may in addition be resources that are both critical and confidential. We call those *internal* resources. Examples of critical resources are the page tables of a hypervisor, the executable code of the untrusted software in a run-time monitor, and in general the resources used by the invariants needed for the verification of functional correctness. Confidential (internal) resources can be cryptographic keys, internal kernel data structures, or the memory of a guest colocated with the application.

The goal is to repair the formal analysis of the kernel, reusing as much as possible of the prior analysis. In particular, our goals are:

1. To demonstrate that critical and internal resources cannot be directly affected by the application and that for these resources the actual system behaves according to the formal specification (i.e. that sequential consistency is preserved and the integrity attacks described in Section C.3.2 cannot succeed).
2. To guarantee that no side channel is present due to caches, i.e. that the real system exposes all and only the channels that are present in the formal functional specification that have been used to verify the kernel using the formal model.

C.6.1 Repairing the Integrity Verification

For simplicity, we assume that the kernel accesses all resources using cacheable virtual addresses. To preserve integrity we must ensure two properties:

- That an address belonging to a critical resource cannot be directly or indirectly modified by the application.
- Sequential consistency of the kernel.

The latter property is equivalent to guaranteeing that what is observed in presence of caches is exactly what is predicted by the ISA specification.

The verification depends on a system invariant that must be preserved by all executions: For every address that belongs to the critical and internal resources, if there is a cache hit and the corresponding cache line differs from the main memory then the cache line must be dirty. The mechanism used to establish this invariant

²In case of Prosper, augmented with a detailed model of the MMU [157].

depends on the specific countermeasure used. It is obvious that if the caches are disabled (Section C.5.1) the invariant holds, since the caches are always empty. In the case of “Always Cacheable Memory” (Section C.5.2) the invariant is preserved because no non-cacheable alias is used to access these resources: the content of the cache can differ from the content of the memory only due to a memory update that changed the cache, thus the corresponding cache line is dirty. Similar arguments apply to the $C \oplus U$ Policy, taking into account that the cache is cleaned whenever a resource type switch from cacheable (C) to uncacheable (U) and vice versa.

More complex reasoning is necessary for other countermeasures, where the attacker can build uncacheable aliases in its own memory. In this case we know that the system is configured so that the application cannot write the critical resources, since otherwise the integrity property cannot be established for the formal model in the first place. Thus, if the cache contains critical or internal data different from main memory it must have been written there by the kernel that only uses cacheable memory only, hence the line is dirty as well.

To show that a physical address pa belonging to a critical resource cannot not be directly or indirectly modified by the application we proceed as follows. By the assumed formal verification, the application has no direct writable access to pa , otherwise the integrity property would not have been established at the ISA level. Then, the untrusted application can not directly update pa neither in the cache nor in the memory. The mechanism that can be used to indirectly update the view of the kernel of the address pa consists in evicting a cache line that has a value for pa different from the one stored in the memory and that is not dirty. However, this case is prevented by the new invariant.

Proving that sequential consistency of the kernel is preserved is trivial: The kernel always uses cacheable addresses so it is unable to break the new invariant: a memory write always updates the cache line if there is a cache hit.

C.6.2 Repairing the Confidentiality Verification

Section C.3 demonstrates the capabilities of the attacker: Additionally to the resources that can be accessed in the formal model (registers, memory locations access to which is granted by the MMU configuration, etc) the attacker is able to measure which cache lines are evicted. Then the attacker can (indirectly) observe all the resources that can affect the eviction. Identifying this set of resources is critical to identify the constraints that must be satisfied by the trusted kernel. For this reason, approximating this set (e.g. by making the entire cache observable) can strongly reduce the freedom of the trusted code. A more refined (still conservative) analysis considers observable by the attacker the cache line tag³ and whether a cache line is empty (cache line emptiness). Then to guarantee confidentiality it

³On direct mapped caches, we can disregard the line tag, because they contain only one way for each line. In order to observe the tags of addresses accessed by the kernel, the attacker requires at least two ways per cache line: one that contains an address accessible by the kernel and one that the attacker can prime in order to measure whether the first line has been accessed.

is necessary to ensure that, while the application is executing, the cache line tag and emptiness never depend on the confidential resources. We stress that this is a sufficient condition to guarantee that no additional information is leaked due to presence of caches with respect to the formal model

Showing that the condition is met by execution of the application is trivial. By the assumed formal verification we already know that the application has no direct read access (e.g. through a virtual memory mapping) to confidential resources. On the other hand, the kernel is able to access these resources, for example to perform encryption. The goal is to show that the caches do not introduce any channel that has not been taken into account at the level of the formal model. Due to the overapproximation described above, this task is reduced to a “cache-state non-interference property”, i.e. showing that if an arbitrary functionality of the kernel is executed then the cache line emptiness and the line tags in the final state do not depend on confidential data.

The analysis of this last verification condition depends on the countermeasure used by the kernel. If the kernel always terminates with caches empty, then the non-interference property trivially holds, since a constant value can not carry any sensible information. This is the case if the kernel always flushes the caches before exiting, never use cacheable aliases (for both program counter and memory accesses) or the caches are completely disabled.

In other cases (e.g. “Secret-Independent Memory Accesses” and “Selective Eviction”) the verification condition is further decomposed to two tasks:

1. Showing that starting from two states that have the same cache states, if two programs access at the same time the same memory locations then the final states have the same cache states.
2. Showing that the sequence of memory accesses performed by the kernel only depends on values that are not confidential.

The first property is purely architectural and thus independent of the kernel. Hereafter we summarise the reasoning for a system with a single level of caches, with separated instruction and data caches and whose caches are physically indexed and physically tagged (e.g. the L1 memory subsystem of ARMv7 CPUs). We use $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2$ to range over machine states and $\sigma_1 \rightarrow \sigma'_1$ to represent the execution of a single instruction. From an execution $\sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n$ we define two projections: $\pi_I(\sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n)$ is the list of encountered program counters and $\pi_D(\sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n)$ is the list of executed memory operations (type of operation and physical address). We define \mathcal{P} as the biggest relation such that if $\sigma_1 \mathcal{P} \sigma_2$ then for both data and instruction cache

- a line in the cache of σ_1 is empty if and only if the same line is empty in σ_2 , and
- the caches of σ_1 and σ_2 have the same tags for every line.

The predicate \mathcal{P} is preserved by executions $\sigma_1 \rightarrow \dots$ and $\sigma_2 \rightarrow \dots$ if the corresponding projections are *cache safe*: (i) the instruction tag and index of $\pi_I(\sigma_1 \rightarrow \dots)[i]$ is equal to the instruction tag and index of $\pi_I(\sigma_2 \rightarrow \dots)[i]$ (ii) if $\pi_D(\sigma_1 \rightarrow \dots)[i]$ is a read (write) then $\pi_D(\sigma_2 \rightarrow \dots)[i]$ is a read (write) (iii) the cache line tag and index of the address in $\pi_D(\sigma_1 \rightarrow \dots)[i]$ is equal to the cache line tag and index of the address in $\pi_D(\sigma_2 \rightarrow \dots)[i]$

Consider the example in Figure C.1a, where va_3 and va_4 are different addresses. In our current setting this is secure only if va_3 and va_4 share the same data cache index and tag (but they could point to different positions within a line). Similarly, the example in Figure C.3 is secure only if the addresses of both targets of the conditional branch have the same instruction cache index and tag. Notice that these conditions are less restrictive than the ones imposed by the program counter security model. Moreover, these restrictions do not forbid completely data-dependent look-up tables. For example, the scrambled implementation of AES presented in Section C.5.3 satisfies the rules that we identified even if it uses data-dependent look-up tables.

In practice, to show that the trusted code satisfies the cache safety policy, we rely on a relational observation equivalence and we use existing tools for relational verification that support trace based observations. In our experiments we adapted the tool presented in [24]. The tool executes two analyses of the code. The first analysis handles the instruction cache: we make every instruction observable and we require that the matched instructions have the same set index and tag for the program counter. The second analysis handles the data cache: we make every memory access an observation and we require that the matched memory accesses use the same set index and tag (originally the tool considered observable only memory writes and required that the matched memory writes access the same address and store the same value). Note that the computation of set index and tag are platform-dependent, thus when porting the same verified code to a processor, whose caches use a different method for indexing lines, the code might not be cache safe anymore. To demonstrate the feasibility of our approach we applied the tool to one functionality of the hypervisor described in Section C.4.2, which is implemented by 60 lines of assembly and whose analysis required 183 seconds.

C.7 Related Work

Kocher [134] and Kelsey et al. [125] were the first to demonstrate cache-based side-channels. They showed that these channels contain enough information to enable an attacker to extract the secret key of cryptographic algorithms. Later, Page formally studied cache side-channels and showed how one can use them to attack naïve implementations of the DES cryptosystem [163]. Among the existing cache attacks, the trace-driven and access-driven attacks are the most closely related to this paper since they can be reproduced using the vectors presented in Section C.3.

In trace-driven attacks [163] an adversary profiles the cache activities while the

victim is executed. Aciçmez showed a trace-driven cache attack on the first two rounds of AES [6], which has been later improved and extended by X. Zhao [234] to compromise a CLEFIA block cipher. A similar result is reported in [36]. In an access-driven, or Prime+Probe, attack the adversary can determine the cache sets modified by the victim. In several papers this technique is used to compromise real cryptographic algorithms like RSA [166, 118] and AES [102, 158, 208].

Due to the security concerns related to cache channels, research on the security implications of shared caches has so far been focusing on padding [231] and mitigation [7] techniques to address timing channels. Notably, Godfrey and Zulkernine have proposed efficient host-based solutions to close timing channels through selective flushing and cache partitioning [90]. In the STEALTHMEM approach [129] each guest is given exclusive access to a small portion of the shared cache for its security critical computations. By ensuring that this stealth memory is always allocated in the cache, no timing differences are observable to an attacker.

In literature, few works investigated cache based storage channels. In fact, all implementations of the above attacks use timing channels as the attack vector. Brumley [44] recently conjectured the existence of a storage channel that can be implemented using cache debug functionality on some ARM embedded microprocessors. However, the ARM technical specification [62] explicitly states that such debug instructions can be executed only by privileged software in TrustZone, making practically impossible for an attacker to access them with the exception of a faulty hardware implementation.

The attack based on mismatched cacheability attributes opens up for TOCT-TOU like vulnerabilities. Watson [223] demonstrated this vulnerability for Linux system call wrappers. A similar approach is used in [43] to invalidate security guarantees, attestation of a platform’s software, provided by a Trusted Platform Module (TPM). TPM takes integrity measurements only before software is loaded into the memory, and it assumes that once the software is loaded it remains unchanged. However, this assumption is not met if the attacker can indirectly change the software before it is used.

Cache-related architectural problems have been exploited before to bypass memory protection. In [226, 78] the authors use a weakness of some Intel x86 implementations to bypass SMRAM protection and execute malicious code in System Management Mode (SMM). The attack relies on the fact that the SMRAM protection is implemented by the memory controller, which is external to the CPU cache. A malicious operating system first marks the SMRAM memory region as cacheable and write-back, then it writes to the physical addresses of the SMRAM. Since the cache is unaware of the SMRAM configuration, the writes are cached and do not raise exceptions. When the execution is transferred to SMM, the CPU fetches the instructions from the poisoned cache. While this work shows similarities to the integrity threat posed by cache storage channels, the above attack is specific to certain Intel implementations and targets only the highest security level of x86. On ARM, the cache keeps track which lines have been filled due to accesses performed by TrustZone SW. The TrustZone SW can configure via its page tables the memory

regions that are considered “secure” (e.g. where its code and internal data structure are stored). A TrustZone access to a secure memory location can hit a cache line only if it belongs to TrustZone.

The attack vectors for data caches presented in this paper abuse undefined behaviour in the ISA specification (i.e., accessing the same memory address with different cacheability types) and deterministic behaviour of the underlying hardware (i.e., that non-cacheable accesses completely bypass the data caches and unexpected cache hits are ignored). While we focused on an ARMv7 processor here, there is a strong suspicion that other architectures exhibit similar behaviour. In fact, in experiments we succeeded to replicate the behaviour of the memory subsystem on an ARMv8 processor (Cortex-A53), i.e., uncacheable accesses do not hit valid entries in the data cache. For Intel x64, the reference manual states that memory type aliases using the page tables and page attribute table (PAT) “may lead to undefined operations that can result in a system failure” ([119], Vol. 3, 11.12.4). It is also explicitly stated that the accesses using the (non-cacheable) WC memory type may not check the caches. Hence, a similar behaviour as on ARM processors should be expected. On the other hand, some Intel processors provide a self-snooping mechanism to support changing the cacheability type of pages without requiring cache flushes. It seems to be similar in effect as the hardware countermeasure suggested in Section C.5.4. In the Power ISA manual ([169], 5.8.2), memory types are assumed to be unique for all aliases of a given address. Nevertheless this is a software condition that is not enforced by the architecture. When changing the storage control bits in page table entries the programmer is required to flush the caches. This also hints to the point that no hardware mechanisms are mandated to handle unexpected cache hits.

Recently, several works successfully verified low level execution platforms that provide trustworthy mechanisms to isolate commodity software. In this context caches are mostly excluded from the analysis. An exception is the work by Barthe et al. [28] that provide an abstract model of cache behaviour sufficient to replicate various timing-based exploits and countermeasures from the literature such as STEALTHMEM.

The verification of seL4 assumes that caches are correctly handled [132] and ignores timing channels. The bandwidth of timing channels in seL4 and possible countermeasures were examined by Cock et al [56]. While storage based channels have not been addressed, integrity of the kernel seems in practice to be preserved by the fact that system call arguments are passed through registers only.

The VerisoftXT project targeted the verification of Microsoft Hyper V and a semantic stack was devised to underpin the code verification with the VCC tool [58]. Guests are modelled as full x64 machines where caches cannot be made transparent if the same address is accessed in cacheable and uncacheable mode, however no implications on security have been discussed. Since the hypervisor uses a shadow page algorithm, where guest translations are concatenated with a secure host translation, the integrity properties do not seem to be jeopardised by any actions of the guest.

Similarly the Nova [199, 203] and CertiKOS [97] microvisors do not consider caches in their formal analysis, but they use hardware which supports a second level address translation which is controlled by the host and cannot be affected by the guest. Nevertheless the CertiKOS system keeps a partition management software in a separate partition that can be contacted by other guests via IPC to request access to resources. This IPC interface is clearly a possible target for attacks using uncacheable aliases.

In any case all of the aforementioned systems seem to be vulnerable to cache storage channel information leakage, assuming they allow the guest systems to set up uncacheable memory mappings. In order to be sound, any proof of information flow properties then needs to take the caches into account. In this paper we show for the first time how to conduct such a non-interference proof that treats also possible data cache storage channels.

C.8 Concluding Remarks

We presented novel cache based attack vectors that use storage channels and we demonstrated their usage to threaten integrity and confidentiality of real software. To the best of our knowledge, it is the first time that cache-based storage channels are demonstrated on commodity hardware.

The new attack vectors partially invalidate the results of formal verification performed at the ISA level. In fact, using storage-channels, the adversary can extract information without accessing variables that are external to the ISA specification. This is not the case for timing attacks and power consumption attacks. For this reason it is important to provide methodologies to fix the existing verification efforts. We show that for some of the existing countermeasures this task can be reduced to checking relational observation equivalence. To make this analysis practical, we adapted an existing tool [24] to check the conditions that are sufficient to prevent information leakage due to the new cache-channels. In general, the additional checks in the code verification need to be complemented by a correctness proof of a given countermeasure on a suitable cache model. In particular it has to be shown that memory coherency for the verified code is preserved by the countermeasure and that an attacker cannot observe sensitive information even if it can create non-cacheable aliases.

The attack presented in Section C.3.2 raises particular concerns, since it poses integrity threats that cannot be carried out using timing channels. The possible victims of such an attack are systems where the ownership of memory is transferred from the untrusted agent to the trusted one and where the trusted agent checks the content of this memory before using it as parameter of a critical function. After that the ownership is transferred, if the cache is not clean, the trusted agent may validate stale input while the critical function uses different data. The practice of transferring ownership between security domains is usually employed to reduce memory copies and is used for example by hypervisors that use direct paging, run-

time monitors that inspect executable code to prevent execution of malware, as well as reference monitors that inspect the content of IP packets or validate requests for device drivers.

There are several issues we leave out as future work. We did not provide a mechanism to check the security of some of the countermeasures like Cache Normalisation and we did not apply the methodology that we described to a complete software. Moreover, the channels that we identified probably do not cover all the existing storage side channels. Branch prediction, TLBs, shareability attributes are all architectural details that, if misconfigured, can lead to behaviours that are inconsistent with the ISA specification. If the adversary is capable of configuring these resources, like in virtualized environments, it is important to identify under which conditions the trusted software preserves its security properties.

From a practical point of view, we focused our experiments on exploiting the L1 cache. For example, to extract the secret key of the AES service on Raspberry Pi 2 we have been forced to flush and clean the L2 cache. The reason is that on this platform the L2 cache is shared with the GPU and we have little to no knowledge about the memory accesses it performs. On the other hand, shared L2 caches open to the experimentation with concurrent attacks, where the attacker can use a shader executed on the GPU. Similarly, here we only treated cache channels on a single processor core. Nevertheless the same channels can be built in a multi-core settings using the shared caches (e.g. L2 on Raspberry Pi 2). The new vectors can then be used to replicate known timing attacks on shared caches (e.g. [118]).

Paper D

Formal Analysis of Countermeasures against Cache Storage Side Channels

D

Hamed Nemati, Roberto Guanciale, Christoph Baumann, Mads Dam

Abstract

Formal verification of systems-level software such as hypervisors and operating systems can enhance system trustworthiness. However, without taking low level features like caches into account the verification may become unsound. While this is a well-known fact w.r.t. timing leaks, few works have addressed latent cache storage side-channels. We present a verification methodology to analyse soundness of countermeasures used to neutralise cache storage channels. We apply the proposed methodology to existing countermeasures, showing that they allow to restore integrity and prove confidentiality of the system. We decompose the proof effort into verification conditions that allow for an easy adaption of our strategy to various software and hardware platforms. As case study, we extend the verification of an existing hypervisor whose integrity can be tampered using cache storage channels. We used the HOL4 theorem prover to validate our security analysis, applying the verification methodology to formal models of ARMv7 and ARMv8.

D.1 Introduction

Formal verification of low-level software such as microkernels, hypervisors, and drivers has made big strides in recent years [3, 10, 225, 71, 109, 233, 199, 97]. We appear to be approaching the point where the promise of provably secure, practical system software is becoming a reality. However, existing verification uses models that are far simpler than contemporary state-of-the-art hardware. Many

features pose significant challenges: Memory models, pipelines, speculation, out-of-order execution, peripherals, and various coprocessors, for instance for system management. In a security context, caches are notorious. They have been known for years to give rise to timing side channels that are difficult to fully counteract [88]. Also, cache management is closely tied to memory management, which—since it governs memory mapping, access control, and cache configuration through page-tables residing in memory—is one of the most complex and security-critical components in the computer architecture flora.

Computer architects strive to hide this complexity from application programmers, but system software, device drivers, and high-performance software, for which tuning of cache usage is critical, need explicit control over features like cacheability attributes. In virtualization scenarios, for instance, it is critical for performance to be able to delegate cache management authority for pages belonging to a guest OS to the guest itself. With such a delegated authority a guest is free to configure its share of the memory system as it wishes, including configurations that may break conventions normally expected for a well-behaved OS. For instance, a guest OS will usually be able to create memory aliases and to set cacheability attributes as it wishes. Put together, these capabilities can, however, give rise to memory incoherence, since the same physical address can now be pointed to by two virtual addresses, one to cache and one to memory. This opens up for cache storage attacks on both confidentiality and integrity, as was shown in [100]. Similarly to cache timing channels that use variations in execution time to discover hardware hidden state, storage channels use aliasing to profile cache activities and to attack system confidentiality. However, while the timing channels are external to models used for formal analysis and do not invalidate verification of integrity properties, storage channels simply make the models unsound: Using them for security analysis can lead to conclusions that are false.

This shows the need to develop verification frameworks for low-level system software that are able to adequately reflect the presence of caches. It is particularly desirable if this can be done in a manner that allows to reuse existing verification tools on simpler models that do not consider caches. This is the goal we set ourselves in this paper. We augment an existing cacheless model by adding a cache and accompanying cache management functionality in MMU and page-tables. We use this augmented model to derive proof obligations that can be imposed to ensure absence of both integrity and confidentiality attacks. This provides a verification framework that we use to analyse soundness of countermeasures. The countermeasures are formally modelled as new proof obligations that can be analysed on the cacheless model to ensure absence of vulnerabilities due to cache storage channels. Since these obligations can be verified using the cacheless model, existing tools [24, 47, 196] (mostly not available on a cache enabled model) can automate this task to a large extent. We then complete the paper by repairing the verification of an existing and vulnerable hypervisor [101], sketching how the derived proof obligations are discharged.

D.2 Related Work

Formal Verification Existing work on formal verification do not takes into account cache storage channels. The verification of seL4 assumes a sequential memory model and leaves cache issues to be managed by means external to model [133, 132]. Cock et al [56] examined the bandwidth of timing channels in seL4 and possible countermeasures including cache coloring. The verification of the Prosper kernel [71, 101] assumes that caches are invisible and correctly handled. Similarly Barthe et al. [29] ignores caches for the verification of an isolation property for an idealised hypervisor. Later, in [30] the authors extended the model to include an abstract account of caches and verified that timing channels are neutralised by cache flushing. The CVM framework [10] treats caches only in the context of device management [110]. Similarly, the Nova [199, 203, 34] and CertiKOS [99, 98, 97] microvisors do not consider caches in their formal analysis. In a follow-up paper [65] the verification was extended to machine code level, using a sequential memory model and relying on the absence of address aliasing.

In scenarios such as OS virtualization, where untrusted software is allowed to configure cacheability of its own memory, all of the above systems can be vulnerable to cache storage channel attacks. For instance, these channels can be used to create illicit information flows among threads of seL4.

Timing Channels Timing attacks and countermeasures have been formally verified to varying degrees of detail in the literature. Almeida et al. [12] prove functional correctness and leakage security of MEE-CBC in presence of timing attackers. Barthe et al. [28] provide an abstract model of cache behaviour sufficient to replicate various timing-based exploits and countermeasures from the literature such as STEALTHMEM [129]. In a follow-up work Barthe et al. [27] formally showed that cryptographic algorithms that are implemented based-on STEALTHMEM approach and thus are S-constant-time¹ are protected against cache-timing attacks. FlowTracker [194] detects leaks using an information flow analysis at compile time. Similarly, Ford et al. [97] uses information flow analysis based on explicit labeling to detect side-channels, and Vieira uses a deductive formal approach to guarantee that side-channel countermeasures are correctly deployed [215]. Other related work includes those adopting formal analysis to either check the rigour of countermeasures [77, 100, 207] or to examine bandwidth of side-channels [135, 76]. Zhou [236] proposed page access based solutions to mitigate the access-driven cache attacks and used model checking to show these countermeasure restore security. Illicit information flows due to caches can also be countered by masking timing fluctuations by noise injection [221] or by clock manipulation [115, 212, 170]. A extensive list of protective means for timing attacks is given in [224, 89].

By contrast, we tackle storage channels. These channels carry information through memory and, additionally to permit illicit information flows, can be used to

¹An implementation is called S-constant-time, if it does not branch on secrets and only memory accesses to stealth addresses can be secret-dependent.

```

V1) D = access(va1)
A1) write(va2,1);
     free(va2)
V2) D = access(va1)
V3) if not policy(D)
     reject
     [evict va1]
V4) use(va1)

```

(a) Integrity

```

A1) write(va1, 1)
A2) write(va2, 0)
V1) if secret
     access(va3)
     else
         access(va4)
A3) D = access(va2)

```

(b) Confidentiality

Figure D.1: Mismatched memory attribute threats

compromise integrity. Storage channels have been used in [100] to show how cache management features could be used to attack both integrity and confidentiality of several types of application.

D.3 Threats and Countermeasures

The presence of caches and ability to configure cacheability of virtual alias enable the class of attacks called “alias-driven attacks” [100]. These attacks are based on building virtual aliases with mismatched cacheability attributes to break memory coherence; i.e, causing inconsistency between the values stored in a memory location and the corresponding cache line, without making the cache line dirty. We present here two examples to demonstrate how integrity and confidentiality can be attacked using this vector.

D.3.1 Integrity

Figure D.1a demonstrates an integrity threat. Here, we assume the cache is direct-mapped, physically indexed and write-back. Also, both the attacker and victim are executed interleaved on a single core. Virtual addresses va_1 and va_2 are aliasing the same memory pa , va_1 is cacheable and va_2 is uncacheable. Initially, the memory pa contains the value 0 and the corresponding cache line is empty. In a sequential model reads and writes are executed in order and their effects are instantly visible: V1) a victim accesses va_1 , reading 0; A1) the attacker writes 1 into pa using va_2 and releases the alias va_2 ; V2) the victim accesses again va_1 , this time reading 1; V3) if 1 does not respect a security policy, then the victim rejects it; otherwise V4) the victim passes 1 to a security-critical functionality.

On a CPU with a weaker memory model the same code behaves differently: V1) using va_1 , the victim reads 0 from the memory and fills the cache; A1) the attacker uses va_2 to directly write 1 in memory, bypasses the cache, and then frees the mapping; V2) the victim accesses again va_1 , reading 0 from the cache; V3) the security policy is evaluated based on 0; possibly, the cache line is evicted and, since

it is not dirty, the memory is not affected; V4) next time the victim accesses pa it reads 1, but 1 is not the value that has been checked against the security policy. This permits the attacker to bypass the policy.²

Intuitive countermeasures against alias-driven attacks are to forbid the attacker from allocating cacheable aliases at all or to make cacheable its entire memory. A lightweight specialization of these approaches is “always cacheability”: A fixed region of memory is made always cacheable and the victim rejects any input pointing outside this region. Coherency can also be achieved by flushing the entire cache before the victim accesses the attacker memory. Unfortunately, this countermeasure comes with severe performance penalties [100]. “Selective eviction” is a more efficient solution and consists in removing from the cache every location that is accessed by the victim and that has been previously accessed by the attacker. Alternatively, the victim can use mismatched cache attributes itself to detect memory incoherence and abort dangerous requests.

D.3.2 Confidentiality

Figure D.1b shows a confidentiality threat. Both va_1 and va_2 point to the location pa and say idx is the cache line index of pa . All virtual addresses except va_2 are cacheable, and we assume that both pa and the physical address pointed by va_3 are allocated in the same cache line. The attacker writes A1) 1 in the cache, making the line dirty, and A2) 0 in the memory. From this point, the value stored in the memory after the execution of the victim depends on the victim behaviour; if the victim accesses at least one address (e.g. va_3) whose line index is idx , then the dirty line is evicted and 1 is written back to the memory; otherwise the line is not evicted and pa still contains 0. This allows the attacker to measure evicted lines and thus to launch an access-driven attack. In the following we summarise some of the countermeasures against cache storage channels presented in [100] and relevant to our work.

Similarly, to ensure that no secret can leak through the cache storage channel, one can forbid allocating cacheable aliases or always flush the cache after executing victim functionalities. An alternative is cache partitioning, where each process gets a dedicated part of the cache and there is no intersection between any two partitions. This makes it impossible for the victim activities to affect the attacker behaviour, thus preventing the attacker to infer information about victim’s internal variables. A further countermeasure is secret independent memory accesses, which aims at transforming the victim’s code so that the victim accesses do not depend on secret.

Cache normalisation can also be used to close storage channels. In this approach, the cache is brought to a known state by reading a sequence of memory cells. This

²Note that the attacker release its alias va_2 before returning the control to the victim, making this attack different from the standard double mapping attacks.

guarantees the subsequent secret dependent accesses only hit the normalised cache lines, preventing the attacker from observing access patterns of the victim.

D.4 High-Level Security Properties

In this work we consider a trusted system software (the “kernel”) that shares the system with an untrusted software (the “application”). Possible instances for the kernel include hypervisors, runtime monitors, low-level operating system routines, and cryptographic services. The application is a software that requests services from the kernel and can be a user process or even a complete operating system. The hardware execution mode used by the application is less privileged than the mode use by the kernel. The application is potentially malicious and takes the role of the attacker here.

Some system resources are owned by the kernel and are called “critical”, some other resources should not be disclosed to the application and are called “confidential”. The kernel dynamically tracks memory ownership and provides mechanisms for secure ownership transfer. This enables the application to pass data to the kernel services, while avoiding expensive copy operations: the application prepares the input inside its own memory, the ownership of this memory is transferred to the kernel, and the corresponding kernel routine operates on the input in-place. Two instances of this are the direct-paging memory virtualization mechanism introduced by Xen [26] and runtime monitors that forbid self-modifying code and prevent execution of unsigned code [51]. In these cases, predictability of the kernel behaviour must be ensured, regardless of any incoherent memory configuration created by the application.

In such a setting, a common approach to formalise security is via an integrity and a confidentiality property. We use $\sigma \in \Sigma$ to represent a state of the system and \rightsquigarrow to denote a transition relation. The transition relation models the execution of one instruction by the application or the execution of a complete handler of the kernel. The integrity property ensures functional correctness (by showing that a state invariant I is preserved by all transitions) and that the critical resources can not be modified by the application (by showing a relation ψ):

Property 10 (Correctness). *For all σ if $I(\sigma)$ and $\sigma \rightsquigarrow \sigma'$ then $I(\sigma')$ and $\psi(\sigma, \sigma')$*

The confidentiality property ensures that confidential resources are not leaked to the application and is expressed using standard non-interference. Let \mathcal{O} be the application’s observations (i.e., the resources that are not confidential) and let $\sim_{\mathcal{O}}$ be observational equivalence (which requires states to have the same observations), then confidentiality is expressed by the following property

Property 11 (Confidentiality). *Let σ_1, σ_2 are initial states of the system such that $\sigma_1 \sim_{\mathcal{O}} \sigma_2$. If $\sigma_1 \rightsquigarrow^* \sigma'_1$ then $\exists \sigma'_2. \sigma_2 \rightsquigarrow^* \sigma'_2$ and $\sigma'_1 \sim_{\mathcal{O}} \sigma'_2$*

The ability of the application to configure cacheability of its resources can lead to incoherency, making formal program analysis on a cacheless model unsound. Nevertheless, directly verifying properties 10 and 11 using a complete cache-aware model is unfeasible for any software of meaningful size. Our goal is to show that the countermeasures can be used to restore coherency. We demonstrate that if the countermeasures are correctly implemented by the kernel then verification of the security properties on the cache-aware model can be soundly reduced to proof obligations on the cacheless model.

D.5 Formalisation

As basis for our study we define two models, a cacheless and a cache-aware model. The cacheless model represents a memory coherent single-core system where all caches are disabled. The cache-aware model is the same system augmented by a single level data cache.

D.5.1 Cacheless Model

The cacheless model is ARM-flavoured but general enough to apply to other architectures. A state $\sigma = \langle reg, psrs, coreg, mem \rangle \in \Sigma$ is a tuple of general-purpose registers reg (including program counter pc), control registers $psrs$, coprocessor state $coreg$, and memory mem . The core executes either in non-privileged mode U or privileged mode P , $Mode(\sigma) \in \{U, P\}$. The control registers $psrs$ encode the execution mode and other execution parameters such as the arithmetic flags. The coprocessor state $coreg$ determines a range of system configuration parameters. The word addressable memory is represented by $mem : \mathbb{PA} \rightarrow \mathbb{B}^w$, where $\mathbb{B} = \{0, 1\}$, \mathbb{PA} be the sets of physical addresses, and w is the word size.

The set \mathbb{R} identifies all resources in the system, including registers, control registers, coprocessor states and physical memory locations (i.e. $\mathbb{PA} \subseteq \mathbb{R}$). We use $Cv : \Sigma \times \mathbb{R} \rightarrow \mathbb{B}^*$ to represent the *core-view* of a resource, which looks up the resource and yields the corresponding value; e.g., for a physical address $pa \in \mathbb{PA}$, Cv returns the memory content in pa , $Cv(\sigma, pa) = \sigma.mem(pa)$.

All software activities are restricted by a hardware monitor. The monitor configuration can depend on coprocessor states (e.g., control registers for a TrustZone memory controller) and on regions of memory (e.g., page-tables for a Memory Management Unit (MMU)). We use the predicate $Mon(\sigma, r, m, acc) \in \mathbb{B}$ to represent the hardware monitor, which holds if in the state σ the access $acc \in \{wt, rd\}$ (for write and read, respectively) to the resource $r \in \mathbb{R}$ is granted for the execution mode $m \in \{U, P\}$. In ARM the hardware monitor consists of a static and a dynamic part. The static access control is defined by the processor architecture, which prevents non-privileged modifications to coprocessor states and control registers and whose model is trivial. The dynamic part is defined by the MMU and controls memory accesses. We use $Mmu(\sigma, va, m, acc) \in (\mathbb{PA} \times \mathbb{B}) \cup \{\perp\}$ to model the memory management unit. This function yields for a virtual address va the translation and the

cacheability attribute if the access permission is granted and \perp otherwise. Therefore, $Mon(\sigma, pa, m, acc)$ is defined as $\exists va. Mmu(\sigma, va, m, acc) = (pa, -)$. Further, $MD : \Sigma \rightarrow \mathbb{R}$ is the function determining resources (i.e., the coprocessor registers, the current master page-table, the linked page-tables) which affect the monitor's behaviour.

The behaviour of the system is defined by an LTS $\rightarrow_m \subseteq \Sigma \times \Sigma$, where $m \in \{U, P\}$ and if $\sigma \rightarrow_m \sigma'$ then $Mode(\sigma) = m$. Each transition represents the execution of a single instruction. When needed, we let $\sigma \rightarrow_m \sigma' [dop]$ denote that the instruction executed has *dop* effects on the data memory subsystem, where *dop* can be $wt(R)$, $rd(R)$, or $cl(R)$ to represent update, read and cache cleaning of resources $R \subseteq \mathbb{R}$. Finally, we use $\sigma_0 \rightsquigarrow \sigma_n$ to represent the weak transition relation that holds if there is a finite execution $\sigma_0 \rightarrow \dots \rightarrow \sigma_n$ such that $Mode(\sigma_n) = U$ and $Mode(\sigma_j) \neq U$ for $0 < j < n$ (i.e., the weak transition hides internal states of the kernel).

D.5.2 Cache-Aware Model

We model a single-core processor with single level unified cache. A state $\bar{\sigma} \in \bar{\Sigma}$ has all the components of the cacheless model together with the cache, $\bar{\sigma} = \langle reg, psrs, coreg, mem, cache \rangle$. The function *Mmu* and transition relation $\rightarrow_m \subseteq \bar{\Sigma} \times \bar{\Sigma}$ are extended to take into account caches. Other definitions of the previous subsection are extended trivially. In the following we use $c\text{-hit}(\bar{\sigma}, pa)$ to denote a cache hit for address *pa*, $c\text{-dirty}(\bar{\sigma}, pa)$ to identify dirtiness of the corresponding cache-line (i.e., if the value of *pa* has been modified in cache but not written back in the memory), and $c\text{-cnt}(\bar{\sigma}, pa)$ to obtain value for *pa* stored in the cache.

Both the kernel and the hardware monitor (i.e., MMU) have the same view of the memory. In fact, the kernel always uses cacheable virtual addresses and the MMU always consults first the cache when it fetches a page table descriptor.³ This allows us to define the core-view for memory resources $pa \in \mathbb{PA}$ in presence of caches:

$$Cv(\bar{\sigma}, pa) = \begin{cases} c\text{-cnt}(\bar{\sigma}, pa) & : c\text{-hit}(\bar{\sigma}, pa) \\ \bar{\sigma}.mem(pa) & : \text{otherwise} \end{cases}$$

D.5.3 Security Properties

Since the application is untrusted, we assume its code is unknown and that it can break its memory coherence at will. Therefore, the behaviour of the application in the cacheless and the cache-aware models can differ significantly. In particular, memory incoherence caused by mismatched cacheability may lead a control variable of the application to have different values in these two models, causing the application to have different control flows. This makes the long-established method of verification by refinement [177] not feasible for analysing behaviour of the attacker.

³This is for instance the case of ARM Cortex-A53 and ARM Cortex-A8.

To accomplish our security analysis we identify a subset of resources that are *critical*: resources for which integrity must be preserved and on which critical functionality depends. The security type of registers, control and coprocessor registers is statically assigned. The security type of memory locations, however, can dynamically change due to transfer of memory ownership; i.e., the type of these resources depends on the state of the system. The function $CR : (\Sigma \cup \Sigma) \rightarrow 2^{\mathbb{R}}$ retrieves the subset of resources that are critical. Function CR usually depends on a subset of resources, for instance the internal kernel data-structures used to store the security type of memory resources. The definition of function CR must be based on the *core-view* to make it usable for both the cacheless and cache-aware models. We also define (for both cacheless and cache-aware state) $Ext_{cr}(\sigma) = \{(a, Cv(\sigma, a)) \mid a \in CR(\sigma)\}$, this is the function that extracts the value of all critical resources. Finally, the set of resources that are confidential is statically defined and identified by $CO \subseteq \mathbb{R}$. This region of memory includes all internal kernel data-structures whose value can depend on secret information.

Property 10 requires to introduce a system invariant \bar{I} that is software-dependent and defined per kernel. The invariant specifies: (i) the shape of a sound page-table (e.g., to prohibit the non-privileged writable accesses to the kernel memory and the page-tables themselves), (ii) properties that permits the kernel to work properly (e.g., the kernel stack pointer and its data structures are correctly configured), (iii) functional properties specific for the selected countermeasure, and (iv) cache-related properties that allow to restore coherency. A corresponding invariant I for the cacheless model is derived from \bar{I} by excluding properties that constrain caches. Property 10 is formally demonstrated by two theorems: one constraining the behaviour of the application and one showing functional correctness of the kernel. Let $ex\text{-entry}(\bar{\sigma})$ be a predicate identifying the state of the system immediately after switching to the kernel (i.e., when an exception handler is executed the mode is privileged and the program counter points to the exception table). Theorem D.5.1 enforces that the execution of the application in the cache enabled setting cannot affect the critical resources.

Theorem D.5.1 (Application-integrity). *For all $\bar{\sigma}$, if $\bar{I}(\bar{\sigma})$ and $\bar{\sigma} \rightarrow_U \bar{\sigma}'$ then $\bar{I}(\bar{\sigma}')$, $Ext_{cr}(\bar{\sigma}) = Ext_{cr}(\bar{\sigma}')$, and if $Mode(\bar{\sigma}') \neq U$ then $ex\text{-entry}(\bar{\sigma}')$*

While characteristics of the application prevents establishing refinement for non-privileged transitions, for the kernel we show that the use of proper countermeasures enables transferring the security properties from the cacheless to the cache-aware model. This demands proving that the two models behave equivalently for kernel transitions. We prove the behavioural equivalence by showing refinement between two models using forward simulation. We define the simulation relation \mathcal{R}_{sim} (guaranteeing equality of critical resources) and show that both the invariant and the relation are preserved by privileged transitions:

Theorem D.5.2 (Kernel-integrity). *For all $\bar{\sigma}_1$ and σ_1 such that $\bar{I}(\bar{\sigma}_1)$, $\bar{\sigma}_1 \mathcal{R}_{sim} \sigma_1$, and $ex\text{-entry}(\bar{\sigma}_1)$ if $\bar{\sigma}_1 \rightsquigarrow \bar{\sigma}_2$ then $\exists \sigma_2$. $\sigma_1 \rightsquigarrow \sigma_2$, $\bar{\sigma}_2 \mathcal{R}_{sim} \sigma_2$ and $\bar{I}(\bar{\sigma}_2)$*

Applying a similar methodology, in Section D.7 we prove the confidentiality property (i.e., Theorem D.5.3) in presence of caches. Here, we use bisimulation (equality of the application’s observations) as unwinding condition:

Theorem D.5.3 (Confidentiality). *For all $\bar{\sigma}_1$ and $\bar{\sigma}_2$ such that $\bar{I}(\bar{\sigma}_1)$, $\bar{I}(\bar{\sigma}_2)$, and $\bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2$, if $\bar{\sigma}_1 \rightsquigarrow \bar{\sigma}'_1$ then $\exists \bar{\sigma}'_2. \bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2$ and $\bar{\sigma}'_1 \sim_{\mathcal{O}} \bar{\sigma}'_2$ as well as $\bar{I}(\bar{\sigma}'_1)$, $\bar{I}(\bar{\sigma}'_2)$.*

D.6 Integrity

Our strategy to demonstrate correctness of integrity countermeasures consists of two steps. We first decompose the proof of Theorems D.5.1 and D.5.2 and show that the integrity properties are met if a set of proof conditions are satisfied. The goal of this step is to provide a theoretical framework that permits to analyse soundness of a countermeasure without the need of dealing with the complex transition relation of the cache-aware model. Then, we demonstrate correctness of two countermeasures of Section D.3.1, namely always cacheability and selective eviction, by showing that if they are correctly implemented by the kernel then verification of the integrity properties can be soundly reduced to analysing properties of the kernel in the cacheless model.

D.6.1 Integrity: Application Level (Theorem D.5.1)

To formalise the proof we introduce the auxiliary definitions of *coherency*, *derivability* and *safety*.

Definition (Coherency). *In $\bar{\sigma} \in \bar{\Sigma}$ a set of memory resources $R \subseteq \mathbb{PA}$ is coherent ($Coh(\bar{\sigma}, R)$), if for all $pa \in R$, such that pa hits the cache and its value differs from the memory (i.e. $\bar{\sigma}.mem(pa) \neq c-cnt(\bar{\sigma}, pa)$), the corresponding cache line is dirty ($c\text{-dirty}(\bar{\sigma}, pa)$).*

Coherency of the critical resources is essential to prove integrity. In fact, for an incoherent resource the *core-view* can be changed indirectly without an explicit memory write, i.e., through evicting the clean cache-line corresponding to the resource which has different values in the cache and memory. Moreover, in some cores, (e.g., ARMv7/v8) the MMU looks first into the caches when it fetches a descriptor. Then if the page-tables are coherent, a cache eviction cannot indirectly affect the behaviour of the MMU.

To allow the analysis of specific countermeasures to abstract from the complex cache-aware transition system, we introduce the notion of *derivability*. This is a relation overapproximating the effects over the memory and cache for instructions executed in non-privileged mode. Derivability is an architectural property and it is independent of the software executing.

Definition (Derivability). *We say $\bar{\sigma}'$ is derivable from $\bar{\sigma}$ in non-privileged mode (denoted as $\bar{\sigma} \triangleright_U \bar{\sigma}'$) if $\bar{\sigma}.coreg = \bar{\sigma}'.coreg$ and for every $pa \in \mathbb{PA}$ one of the following properties holds:*

$$\begin{aligned}
D_{\emptyset}(\bar{\sigma}, \bar{\sigma}', pa) &\stackrel{\text{def}}{=} M'(pa) \neq M(pa) \Rightarrow (c\text{-dirty}(\bar{\sigma}, pa) \wedge M'(pa) = c\text{-cnt}(\bar{\sigma}, pa)) \\
&\wedge W(\bar{\sigma}', pa) \neq W(\bar{\sigma}, pa) \Rightarrow \\
&(\neg c\text{-hit}(\bar{\sigma}', pa) \wedge (c\text{-dirty}(\bar{\sigma}, pa) \Rightarrow M'(pa) = c\text{-cnt}(\bar{\sigma}, pa))) \\
\\
D_{rd}(\bar{\sigma}, \bar{\sigma}', pa) &\stackrel{\text{def}}{=} Mon(\bar{\sigma}, pa, U, rd) \\
&\wedge M'(pa) = M(pa) \wedge W(\bar{\sigma}', pa) \neq W(\bar{\sigma}, pa) \Rightarrow \\
&(c\text{-cnt}(\bar{\sigma}', pa) = M(pa) \wedge \neg c\text{-hit}(\bar{\sigma}, pa)) \\
\\
D_{wt}(\bar{\sigma}, \bar{\sigma}', pa) &\stackrel{\text{def}}{=} Mon(\bar{\sigma}, pa, U, wt) \\
&\wedge (W(\bar{\sigma}', pa) \neq W(\bar{\sigma}, pa) \Rightarrow c\text{-dirty}(\bar{\sigma}', pa)) \\
&\wedge (M'(pa) \neq M(pa) \Rightarrow \\
&(\neg c\text{-dirty}(\bar{\sigma}', pa) \Rightarrow \exists va. Mmu(\bar{\sigma}, va, U, wt) = (pa, 0)))
\end{aligned}$$

Figure D.2: Derivability. Here $M = \bar{\sigma}.mem$, $M' = \bar{\sigma}'.mem$, and $W(\bar{\sigma}, pa) = \langle c\text{-hit}(\bar{\sigma}, pa), c\text{-dirty}(\bar{\sigma}, pa), c\text{-cnt}(\bar{\sigma}, pa) \rangle$ denotes the cache-line corresponding to pa in $\bar{\sigma}.cache$.

$D_{\emptyset}(\bar{\sigma}, \bar{\sigma}', pa)$: *Independently of the access rights for the address pa , the corresponding memory can be changed due to an eviction of a dirty line and subsequent write-back of the cached value into the memory. Moreover, the cache can always change due to an eviction.*

$D_{rd}(\bar{\sigma}, \bar{\sigma}', pa)$: *If non-privileged mode can read the address pa , the cache state can change through a fill operation which loads the cache with the value of pa in the memory.*

$D_{wt}(\bar{\sigma}, \bar{\sigma}', pa)$: *If non-privileged mode can write the address pa , it can either write directly into the cache, making it dirty, or bypass it, by using an uncacheable alias.*

Figure D.2 reports the formal definition of these predicates.

Definition (Safety). A state $\bar{\sigma}$ is safe, $safe(\bar{\sigma})$, if for every state $\bar{\sigma}'$, resource r , mode m and access request acc if $\bar{\sigma} \triangleright_U \bar{\sigma}'$ then $Mon(\bar{\sigma}, r, m, acc) = Mon(\bar{\sigma}', r, m, acc)$.

A state is *safe* if non-privileged executions cannot affect the hardware monitor, i.e., only the kernel can change page-tables.

To decompose proof of Theorem D.5.1, the invariant \bar{I} must be split in three parts: a functional part \bar{I}_{fun} which only depends on the *core-view* for the critical resources, an invariant \bar{I}_{coh} which only depends on coherency of the critical resources, and an optional countermeasure-specific invariant \bar{I}_{cm} which depends on coherency of non-critical memory resources such as resources in an always-cacheable region.

Proof Obligation D.6.1. *For all $\bar{\sigma}$, $\bar{I}(\bar{\sigma}) = \bar{I}_{fun}(\bar{\sigma}) \wedge \bar{I}_{coh}(\bar{\sigma}) \wedge \bar{I}_{cm}(\bar{\sigma})$ and:*

1. *for all $\bar{\sigma}'$ if $Ext_{cr}(\bar{\sigma}) = Ext_{cr}(\bar{\sigma}')$ then $\bar{I}_{fun}(\bar{\sigma}) = \bar{I}_{fun}(\bar{\sigma}');$*
2. *for all $\bar{\sigma}'$ if $Coh(\bar{\sigma}, CR(\bar{\sigma}))$, $Coh(\bar{\sigma}', CR(\bar{\sigma}'))$, and $Ext_{cr}(\bar{\sigma}) = Ext_{cr}(\bar{\sigma}')$ then $\bar{I}_{coh}(\bar{\sigma}) = \bar{I}_{coh}(\bar{\sigma}');$*
3. *for all $\bar{\sigma}'$ if $\bar{I}(\bar{\sigma})$ and $\bar{\sigma} \triangleright_U \bar{\sigma}'$ then $\bar{I}_{cm}(\bar{\sigma}').$*

Also, the function CR must be correctly defined: i.e. resources affecting the set of critical resources are critical themselves.

Proof Obligation D.6.2. *For all $\bar{\sigma}, \bar{\sigma}'$ if $Ext_{cr}(\bar{\sigma}) = Ext_{cr}(\bar{\sigma}')$ then $CR(\bar{\sigma}) = CR(\bar{\sigma}')$*

Safety is essential to prove integrity: if a state is not safe, the application can potentially elevate its permissions by changing configurations of the hardware monitor and get access to resources beyond its rights.

Lemma D.6.1. *If $\bar{I}(\bar{\sigma})$ then $safe(\bar{\sigma})$*

Proof of Lemma D.6.1 depends on the formal model of the hardware monitor and guarantees provided by the invariant. Using the invariant \bar{I} , we identify three main proof obligations that are needed to prove this lemma.

1. The functional part of the invariant must guarantee that the resources that control the hardware monitor are considered critical.

Proof Obligation D.6.3. *If $\bar{I}_{fun}(\bar{\sigma})$ then $MD(\bar{\sigma}) \subseteq CR(\bar{\sigma})$*

2. The application should not be allowed to directly affect the critical resources. This means there is no address writable in non-privileged mode that points to a critical resource.

Proof Obligation D.6.4. *If $\bar{I}_{fun}(\bar{\sigma})$ and $r \in CR(\bar{\sigma})$ then $\neg Mon(\bar{\sigma}, r, U, wt)$*

3. Finally, to prevent the application from indirectly affecting the hardware monitor, e.g., by line eviction, the invariant must ensure coherency of critical resources.

Proof Obligation D.6.5. *If $\bar{I}_{coh}(\bar{\sigma})$ then $Coh(\bar{\sigma}, CR(\bar{\sigma}))$*

We overapproximate the reachable states in non-privileged mode to prove properties that do not depend on the software platform or countermeasure. This eliminates the need for revalidating properties of the instruction set (i.e., properties that must be verified for every possible instruction) in every new software scenario.

Lemma D.6.2. *For all $\bar{\sigma}$ such that $\text{safe}(\bar{\sigma})$ and $\text{Coh}(\bar{\sigma}, \text{MD}(\bar{\sigma}))$, if $\bar{\sigma} \rightarrow_U \bar{\sigma}'$ then⁴*

1. $\bar{\sigma} \triangleright_U \bar{\sigma}'$, i.e. non-privileged transitions from safe states can only lead into derivable states
2. if $\text{Mode}(\bar{\sigma}') \neq U$ then $\text{ex-entry}(\bar{\sigma}')$, i.e., the mode can only change by entering an exception handler

Corollary 3 shows that derivability can be used as sound overapproximation of the behaviour of non-privileged transitions if the invariant holds.

Corollary 3. *For all $\bar{\sigma}$ if $\bar{I}(\bar{\sigma})$ and $\bar{\sigma} \rightarrow_U \bar{\sigma}'$ then $\bar{\sigma} \triangleright_U \bar{\sigma}'$*

Proof. The statement directly follows by Lemma D.6.2.1 which is enabled by Lemma D.6.1, Obligation D.6.3, and Obligation D.6.5. \square

We now proceed to show that the hardware monitor enforces access policy correctly; i.e., the application transitions cannot modify critical resources.

Lemma D.6.3. *For all $\bar{\sigma}, \bar{\sigma}'$ such that $\bar{I}(\bar{\sigma})$ if $\bar{\sigma} \triangleright_U \bar{\sigma}'$ then $\text{Ext}_{\text{cr}}(\bar{\sigma}) = \text{Ext}_{\text{cr}}(\bar{\sigma}')$*

Proof. Since $\bar{I}(\bar{\sigma})$ holds, the hardware monitor prohibits writable accesses of the application to critical resources (Obligation D.6.4) and $\text{safe}(\bar{\sigma})$ holds (Lemma D.6.1). Also, derivability shows that the application can directly change only resources that are writable according to the monitor. Thus, the application cannot directly update $\text{CR}(\bar{\sigma})$. Beside, the invariant guarantees coherency of critical resources in $\bar{\sigma}$ (Obligation D.6.5). This prevents indirect modification of these resources. \square

To complete the proof of Theorem D.5.1 we additionally need to show that coherency of critical resources (Lemma D.6.4) and invariant (Lemma D.6.5) are preserved by non-privileged transitions.

Lemma D.6.4. *For all $\bar{\sigma}$ if $\bar{I}(\bar{\sigma})$ and $\bar{\sigma} \triangleright_U \bar{\sigma}'$ then $\text{Coh}(\bar{\sigma}', \text{CR}(\bar{\sigma}'))$*

Proof. The proof depends on Obligation D.6.5 and Obligation D.6.4: coherency can be invalidated only through non-cacheable writes, which are not possible since aliases to critical resources that are writable by non-privileged mode are forbidden. \square

Lemma D.6.5. *For all $\bar{\sigma}$ and $\bar{\sigma}'$ if $\bar{I}(\bar{\sigma})$ and $\bar{\sigma} \rightarrow_U \bar{\sigma}'$ then $\bar{I}_{\text{fun}}(\bar{\sigma}')$*

Proof. To show that non-privileged transitions preserve the invariant we use Corollary 3, Lemma D.6.3, and Obligation D.6.1.1. \square

Finally, Lemma D.6.2.2, Corollary 3, Lemma D.6.3, Lemma D.6.4, Lemma D.6.5, and Obligation D.6.1 imply Theorem D.5.1, completing the proof of integrity for non-privileged transitions.

⁴In [127] the authors proved a similar theorem for the HOL4 ARMv7 model provided by Anthony Fox et. al. [82].

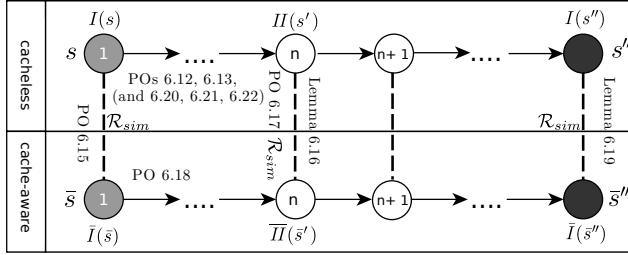


Figure D.3: Verification of kernel integrity

D.6.2 Integrity: Kernel Level (Theorem D.5.2)

The proof of kernel integrity (Theorem D.5.2) is done by: (i) showing that the kernel preserves the invariant I in the cacheless model (Obligation D.6.6); (ii) requiring a set of general proof obligations (e.g., the kernel does not jump outside its address space) that must be verified at the cacheless level (Obligation D.6.7), (iii) demonstrating that the simulation relation permits transferring the invariant from the cache-aware model to the cacheless one (Obligation D.6.8), (iv) verifying a refinement between the cacheless model and the cache-aware model (Lemma D.6.7) assuming correctness of the countermeasure (Obligation 1), and finally (v) proving that the refinement theorem allow to transfer the invariant from the cacheless model to the cache-aware model (Lemma D.6.8). Figure D.3 indicates our approach to prove kernel integrity.

Our goal is to lift the code verification to the cacheless model, so that existing tools can be used to discharge proof obligations. The first proof obligation requires to show that the kernel is functionally correct when there is no cache:

Proof Obligation D.6.6. *For all σ such that $I(\sigma)$ and ex-entry(σ) if $\sigma \rightsquigarrow \sigma'$ then $I(\sigma')$.*

To enable the verification of simulation, it must be guaranteed that the kernel's behaviour is predictable: (D.6.7.1) the kernel cannot execute instructions outside the critical region (this property is usually verified by extracting the control flow graph of the kernel and demonstrating that it is contained in a static region of the memory), and (D.6.7.2) the kernel does not change page-tables entries that maps the kernel virtual memory (here this region is assumed to be static and identified by \mathbf{K}_{vm}) which must be accessed only using cacheable aliases.

Proof Obligation D.6.7. *For all σ such that $I(\sigma)$ and ex-entry(σ) if $\sigma \rightarrow_P^* \sigma'$ then*

1. if $Mmu(\sigma', \sigma'.reg.pc, P, acc) = (pa, -)$ then $pa \in CR(\sigma)$
2. for every $va \in \mathbf{K}_{\text{vm}}$ and acc if $Mmu(\sigma, va, P, acc) = (pa, c)$ then $Mmu(\sigma', va, P, acc) = (pa, c)$ and $c = 1$

In order to establish a refinement, we introduce the *memory-view* Mv of the cache-aware model. This function models what can be observed in the memory after a cache line eviction and is used to handle dynamic growth of the critical resources, which are potentially not coherent when a kernel handler starts.

$$Mv(\bar{\sigma}, r) \stackrel{\text{def}}{=} \begin{cases} c\text{-}cnt(\bar{\sigma}, pa) & : r \in \mathbb{PA} \wedge c\text{-}dirty(\bar{\sigma}, pa) \\ \bar{\sigma}.mem(pa) & : r \in \mathbb{PA} \wedge \neg c\text{-}dirty(\bar{\sigma}, pa) \\ Cv(\bar{\sigma}, r) & : \text{otherwise} \end{cases}$$

Note that Mv only differs from Cv in memory resources that are cached, clean, and have different values stored in the cache and memory, i.e., incoherent memory resources. In particular, we can prove the following lemma about coherency:

Lemma D.6.6. *Given a memory resource $pa \in \mathbb{PA}$ and cache-aware state $\bar{\sigma}$ then $Coh(\bar{\sigma}, \{pa\}) \Leftrightarrow (Cv(\bar{\sigma}, pa) = Mv(\bar{\sigma}, pa))$, i.e., memory resources are coherent, iff both views agree on them.*

We define the simulation relation between the cacheless and cache-aware models using the memory-view: $\bar{\sigma} \mathcal{R}_{sim} \sigma \equiv \forall r \in \mathbb{R}. Cv(\sigma, r) = Mv(\bar{\sigma}, r)$. The functional invariant of the cache-aware model and the invariant of the cacheless model must be defined analogously, that is the two invariants are equivalent if the critical resources are coherent, thus ensuring that functional properties can be transferred between the two models.

Proof Obligation D.6.8. *For all $\bar{\sigma}$ and σ , such that $\bar{I}_{coh}(\bar{\sigma})$ and $\bar{\sigma} \mathcal{R}_{sim} \sigma$, holds $\bar{I}_{fun}(\bar{\sigma}) \Leftrightarrow I(\sigma)$*

A common problem of verifying low-level software is coupling the invariant with every possible internal states of the kernel. This is a major concern here, since the set of critical resources changes dynamically and can be stale while the kernel is executing. We solve this problem by defining a *internal invariant* $\bar{II}(\bar{\sigma}, \bar{\sigma}')$ which allows us to define properties of the state $\bar{\sigma}'$ in relation with the initial state of the kernel handler $\bar{\sigma}$. This invariant \bar{II} (similarly II for the cacheless model) includes: (i) $\bar{I}(\bar{\sigma})$ holds, (ii) the program counter in $\bar{\sigma}'$ points to the kernel memory, (iii) coherency of resources critical in the initial state (i.e. $Coh(\bar{\sigma}', CR(\bar{\sigma}))$) (iv) all virtual addresses in \mathbf{K}_{vm} are cacheable and their mapping in $\bar{\sigma}$ and $\bar{\sigma}'$ is unchanged, and (v) additional requirements that are countermeasure specific and will be described later.

Lemma D.6.7. *For all $\bar{\sigma}$ and σ such that $\bar{I}(\bar{\sigma})$, $ex\text{-}entry(\bar{\sigma})$, and $\bar{\sigma} \mathcal{R}_{sim} \sigma$, if $\bar{\sigma} \rightarrow_P^n \bar{\sigma}'$ and $\sigma \rightarrow_P^n \sigma'$ then $\bar{\sigma}' \mathcal{R}_{sim} \sigma'$, $\bar{II}(\bar{\sigma}, \bar{\sigma}')$, and $II(\sigma, \sigma')$*

Proof. By induction on the execution length. The base case is trivial, since no step is taken. For the inductive case we first show that the instruction executed is the same in both the models: \mathcal{R}_{sim} guarantees that the two states have the same program counter, Obligation D.6.7.1 ensures that the program counter points to

the kernel memory contained in the critical resources, property (iii) of the invariant guarantees that this memory region is coherent, and Lemma D.6.6 shows that the fetched instruction is the same.

Proving that the executed instruction preserves kernel virtual memory mapping is trivial, since Obligation D.6.7.2 ensures that the kernel does not change its own memory layout and that it only uses cacheable aliases. Showing that the resources accessed by the instruction have the same value (thus guaranteeing \mathcal{R}_{sim} is preserved) in the cache-aware and cacheless states depends on demonstrating their coherency. This is countermeasure specific and is guaranteed by proof obligation D.6.9.1. Similarly, showing the internal invariant is maintained by privileged transitions depends on the specific countermeasure that is in place (Obligation D.6.9.2). \square

Proof Obligation D.6.9. *For all states $\bar{\sigma}$ and $\bar{\sigma}'$ that satisfy the refinement (i.e. $\bar{I}(\bar{\sigma})$, ex-entry($\bar{\sigma}$), and $\bar{\sigma} \mathcal{R}_{sim} \sigma$), after any number n of instructions of the kernel that preserve the refinement and the internal invariants ($\bar{\sigma} \rightarrow_P^n \bar{\sigma}'$, $\sigma \rightarrow_P^n \sigma'$, $\bar{\sigma}' \mathcal{R}_{sim} \sigma'$, $\bar{II}(\bar{\sigma}, \bar{\sigma}')$, and $II(\sigma, \sigma')$)*

1. *if the execution of the $n + 1$ -th instruction in the cacheless model accesses resources R ($\sigma' \rightarrow_P \sigma''$ [dop] and either $dop = rd(R)$ or $dop = wt(R)$) then $Coh(\bar{\sigma}', R)$*
2. *the execution of the $n + 1$ -th instruction in the cacheless and cache-aware models ($\sigma' \rightarrow_P \sigma''$ and $\bar{\sigma}' \rightarrow_P \bar{\sigma}''$) preserves the internal invariants ($II(\sigma, \sigma'')$ and $\bar{II}(\bar{\sigma}, \bar{\sigma}'')$)*

Additionally, the internal invariant must ensure the countermeasure specific requirements of coherency for all internal states of the kernel.

Proof Obligation D.6.10. *For all $\bar{\sigma}$ and $\bar{\sigma}'$ if $\bar{II}(\bar{\sigma}, \bar{\sigma}')$ then $\bar{I}_{cm}(\bar{\sigma}')$*

Finally, we show that the invariant \bar{I} holds in a cache-aware state when the control is returned to non-privileged mode, i.e. when the invariant is re-established in the cacheless model.

Lemma D.6.8. *For all $\bar{\sigma}$, $\bar{\sigma}'$, and σ' if $I(\sigma')$, $\bar{\sigma}' \mathcal{R}_{sim} \sigma'$, and $\bar{II}(\bar{\sigma}, \bar{\sigma}')$ then $\bar{I}(\bar{\sigma}')$ holds.*

Proof. The three parts of invariant $\bar{I}(\bar{\sigma}')$ are demonstrated independently. Obligation D.6.10 establishes $\bar{I}_{cm}(\bar{\sigma}')$. Property (iii) of $\bar{II}(\bar{\sigma}, \bar{\sigma}')$ guarantees $\bar{I}_{coh}(\bar{\sigma}')$. Finally, Obligation D.6.8 demonstrates $\bar{I}_{fun}(\bar{\sigma})$. \square

D.6.3 Correctness of countermeasures

Next, we turn to show that selected countermeasures for the integrity attacks prevent usage of cache to violate the integrity property. Thus, we show that the countermeasures help to discharge the coherency related proof obligations reducing

verification of integrity to analysing properties of the kernel code using the cacheless model.

Always Cacheability We use $M_{ac} \subseteq \mathbb{PA}$ to statically identify the region of physical memory that should be always accessed using cacheable aliases. The verification that the always cacheability countermeasure is in place can be done by discharging the following proof obligations at the cacheless level: (D.6.11.1) the hardware monitor does not permit uncacheable accesses to M_{ac} , (D.6.11.2.a) the kernel never allocates critical resources outside M_{ac} , thus restricting the application to use M_{ac} to communicate with the kernel, and (D.6.11.2.b) the kernel accesses only resources in M_{ac} :

Proof Obligation D.6.11. *For all σ such that $I(\sigma)$*

1. *for every va, m and acc if $Mmu(\sigma, va, m, acc) = (pa, c)$ and $pa \in M_{ac}$ then $c = 1$,*
2. *if $\sigma \rightarrow_P^* \sigma'$ then*
 - a) *$CR(\sigma') \subseteq M_{ac}$ and*
 - b) *if $\sigma' \rightarrow_P \sigma''$ [dop] and R are the resources in dop then $R \subseteq M_{ac}$*

These three properties, together with Obligation D.6.8, enable us to prove that the resources accessed by the instructions executed in the privileged mode have the same value in the cache-aware and cacheless states (Obligation D.6.9.1). In a similar vein, we instantiate part (v) of the internal invariant as $Coh(\bar{\sigma}', M_{ac})$, and then we use Obligations D.6.7 and D.6.11.2.a to show that the internal invariant is preserved by kernel steps (Obligation D.6.9.2). To discharge coherency related proof obligations for non-privileged mode, we set \bar{I}_{coh} and \bar{I}_{cm} to be $Coh(\bar{\sigma}, M_{ac} \cap CR(\bar{\sigma}))$ and $Coh(\bar{\sigma}, M_{ac} \setminus CR(\bar{\sigma}))$ respectively. This makes the proof of Obligations D.6.1.2, D.6.5, and D.6.10 trivial.

We use Lemma D.6.1 to guarantee that derivability preserves page-tables and, thus, cacheability of the resources in M_{ac} , and we use Lemma D.6.3 and Obligation D.6.2 to demonstrate that derivability preserves CR . Finally, Obligation D.6.11.1 enforces that all aliases to M_{ac} are cacheable, demonstrating Obligations D.6.1.3.

Selective Eviction This approach requires to selectively flush the lines that correspond to the memory locations that become critical when the kernel acquire ownership of a region of memory. To verify that the kernel correctly implements this countermeasure we need to track evicted lines, by adding to the cacheless model a history variable h .

$$\frac{\sigma \rightarrow_P \sigma' [cl(R)]}{(\sigma, h) \rightarrow_P (\sigma', h \cup R) [cl(R)]} \quad \frac{\sigma \rightarrow_P \sigma' [dop] \wedge dop \neq cl(R)}{(\sigma, h) \rightarrow_P (\sigma', h) [dop]}$$

All the kernel accesses must be restricted to resources that are either critical or have been previously cleaned (i.e. are in h).

Proof Obligation D.6.12. *For all σ such that $I(\sigma)$ if $(\sigma, \emptyset) \rightarrow_P^* (\sigma', h')$, $(\sigma', h') \rightarrow_P (\sigma'', h'')$ [dop], and either $dop = rd(R)$ or $dop = wt(R)$ then $R \subseteq CR(\sigma) \cup h'$*

Moreover, it must be guaranteed that the set of critical resources always remains coherent.

Proof Obligation D.6.13. *For all σ such that $I(\sigma)$ and ex-entry(σ) if $(\sigma, \emptyset) \rightsquigarrow (\sigma', h')$ then $CR(\sigma') \subseteq CR(\sigma) \cup h'$*

This ensures that the kernel accesses only coherent resources and allows to establish Obligation D.6.9.1. To discharge Obligation D.6.9.2, we first define part (v) of the internal invariant as $Coh(\bar{\sigma}', CR(\bar{\sigma}) \cup h')$ and then use Obligations D.6.7 and D.6.12 to discharge it. To discharge proof obligations of non-privileged mode in the cache-aware model, for state $\bar{\sigma}$ the invariant must ensure $\bar{I}_{coh}(\bar{\sigma}) = Coh(\bar{\sigma}, CR(\bar{\sigma}))$ and $\bar{I}_{cm}(\bar{\sigma}) = true$. This makes the proof of Obligations D.6.1.2, D.6.5 and D.6.10 trivial. Part (v) of the internal invariant and Obligation D.6.13 ensure that \bar{I}_{coh} is established when the kernel returns to the application.

D.7 Confidentiality

This section presents the proof of the confidentiality property. The proof relies on the cache behaviour, hence we briefly present the cache model.

D.7.1 Generic Data Cache Model

We have formally defined a generic model which fits a number of common processor data-cache implementations. The intuition behind is that most data-caches are direct mapped or set-associative caches, sharing a similar structure: (1) Memory is partitioned into sets of lines which are congruent w.r.t. to a set index, (2) data-caches contain a number of ways which can hold one corresponding line for every set index, being uniquely identified by a tag, (3) writes can make lines dirty, i.e., potentially different from the associated data in memory, (4) there is a small set of common cache operations, e.g., filling the cache with a line from memory, (5) an eviction policy controls the allocation of ways for new lines, and the eviction of old lines if the cache is full.

In addition to the cache contents, partitioned into line sets, we keep history $H \in \mathbb{A}^*$ of internal cache actions performed for each line set. An action $a \in \mathbb{A}$ can be (1) a read or write access to a present line, (2) a line eviction, or (3) a line fill. All actions also specify the tag of the corresponding line.

As each line set can only store limited amounts of entries, eviction policy $evict?(H, t)$ returns the tag of the line to be replaced at a line fill for a given tag t , or \perp if eviction is not required. Evicted dirty lines are then written back into the memory.

We assume here that the eviction policy is only depending on the action history of a line set and the tag of the line to be filled in. This is trivially the case for

direct-mapped caches, where each line set at most contains one entry and no choice of eviction targets is possible, and for purely random replacements strategies, where the cache state and history is completely ignored. Also more sophisticated eviction policies, like LRU or pseudo-LRU, usually only depend on preceding operations on the same line set.

Another observation for these replacement strategies is that they only depend on finite subsequences of the history. For instance, LRU only depends on the past actions on the entries currently present in a given set and after a cache flush subsequent line fills and evictions are not influenced by operations from before the flush. We formalize this notion as the *filtered history* on which a given eviction policy depends, computed from the full history by policy-specific filter $\varphi : \mathbb{A}^* \rightarrow \mathbb{A}^*$. The idea is that the eviction policy makes the same decisions for filtered and unfiltered histories.

Assumption D.7.1. *If φ is a policy-specific filter then for all $H \in \mathbb{A}^*$ and tag t holds, $\text{evict?}(H, t) = \text{evict?}(\varphi(H), t)$.*

We use this property in combination with our countermeasures in the confidentiality proof, in order to make cache states indistinguishable for the attacker. A complete formalization of the cache state and semantics is omitted here, as these details are irrelevant for the general proof strategy outlined below (see Appendix for the detailed model).

D.7.2 Observations in the Cache-Aware Model

The kernel and untrusted application share and control parts of the memory and the caches. Our goal is to ensure that through these channels the application cannot infer anything about the confidential resources of the kernel. We fix the confidential resources of the kernel as a static set $CO \subset \mathbb{R}$ and demand that they cannot be directly accessed by the application (Obligation D.7.2).

The observations of the application are over-approximated by the set $\mathcal{O} = \{r \mid r \notin CO\}$. Note, that some of the observable resources may be kernel resources that are not directly accessible by the application, but affect its behaviour (e.g., page tables). Two cacheless states σ_1 and σ_2 are considered observationally equivalent, if all resources in \mathcal{O} have the same core-view. Formally we define $\sigma_1 \sim_{\mathcal{O}} \sigma_2 \equiv \forall r \in \mathcal{O}. Cv(\sigma_1, r) = Cv(\sigma_2, r)$.

In the cache-aware model, naturally, also the cache has to be taken into account. Specifically, for addresses readable by the attacker, both the corresponding cache line and underlying main memory content can be extracted using uncacheable aliases. For all other cache lines, we overapproximate the observable state, assuming that the attacker can infer whether a tag is present in the cache (*tag state*⁵),

⁵Tags of kernel accesses can be identified when a line set contains both a line of the attacker and the kernel, and subsequent secret-dependent accesses may either cause a hit on that line or a miss which evicts the attacker's line.

measure whether an entry is dirty, and derive the filtered history⁶ of cache actions on all line sets. For caches and memories C_1 , C_2 , M_1 , and M_2 we denote observational equivalence w.r.t. a set $A \subseteq \mathbb{PA}$ by $(C_1, M_1) \approx_A (C_2, M_2)$. The relation holds if:

1. the memories agree for all $pa \in A$, i.e., $M_1(pa) = M_2(pa)$,
2. the line sets with any index i in C_1 , C_2 :
 - a) agree on the tags of their entries (same tag state),
 - b) agree on the dirtiness of their entries
 - c) agree on the contents of those entries that have tags pointing to addresses in A , and
 - d) agree on their filtered history (“ $\varphi(H_1(i)) = \varphi(H_2(i))$ ”).

Notice that \approx_A implies core-view equivalence for any address in A .

Now we distinguish observable resources which are always coherent, from potentially incoherent non-critical memory resources $NC \subseteq \mathbb{PA} \cap \mathcal{O}$. Intuitively, this set contains all observable addresses that the application may access through uncacheable aliases. For coherent observable resources we introduce relation

$$\bar{\sigma}_1 \sim_{coh} \bar{\sigma}_2 \stackrel{\text{def}}{=} \forall r \in \mathcal{O} \setminus NC. Cv(\bar{\sigma}_1, r) = Cv(\bar{\sigma}_2, r),$$

and define observational equivalence for the cache-aware model:

$$\begin{aligned} \bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2 &\stackrel{\text{def}}{=} \bar{\sigma}_1 \sim_{coh} \bar{\sigma}_2 \\ &\wedge (\bar{\sigma}_1.\text{cache}, \bar{\sigma}_1.\text{mem}) \approx_{NC} (\bar{\sigma}_2.\text{cache}, \bar{\sigma}_2.\text{mem}). \end{aligned}$$

Note that we are overloading notation here and that \sim_{coh} and $\sim_{\mathcal{O}}$ are equivalence relations. Allowing to apply relation \sim_{coh} also to cacheless states, we get the following corollary.

Corollary 4. *For all σ_1, σ_2 , if $\sigma_1 \sim_{\mathcal{O}} \sigma_2$ then $\sigma_1 \sim_{coh} \sigma_2$.*

Confidentiality (Theorem D.5.3) is demonstrated by showing that relation $\sim_{\mathcal{O}}$ is a bisimulation (i.e., it is preserved for pairs of computations starting in observationally equivalent states). Below, we prove this property separately for of application and kernel steps.

⁶The history of kernel cache operations may be leaked in a similar way as the tag state, affecting subsequent evictions in the application, however only as far as the filter for the eviction policy allows it.

D.7.3 Confidentiality: Application Level

As relation $\sim_{\mathcal{O}}$ is countermeasure-independent, verification for application's transition can be shown once and for all for a given hardware platform, assuming the kernel invariant guarantees several properties for the hardware configuration. First, the hardware monitor must ensure that confidential resources are only readable in privileged mode.

Proof Obligation D.7.2. *For all $\bar{\sigma}$ such that $\bar{I}(\bar{\sigma})$ if $r \in CO$ then $\neg Mon(\bar{\sigma}, r, U, rd)$.*

Secondly, the invariant needs to guarantee that the hardware monitor data is always observable. This implies that in observationally equivalent states the same access permissions are in place.

Proof Obligation D.7.3. *For all $\bar{\sigma}$, if $\bar{I}(\bar{\sigma})$ then $MD(\bar{\sigma}) \subset \mathcal{O}$.*

In addition, the monitor never allows the application to access coherent resources through uncacheable aliases.

Proof Obligation D.7.4. *For all $\bar{\sigma}$ such that $\bar{I}(\bar{\sigma})$ for every va , access right acc if $Mmu(\bar{\sigma}, va, U, acc) = (pa, c)$ and $pa \in \mathbb{PA} \setminus NC$ then $c = 1$.*

By this property it is then easy to derive the coherency of resources outside of NC , assuming that the hardware starts in a coherent memory configuration and that the kernel never makes memory incoherent itself (Obligation D.6.7.2).

Proof Obligation D.7.5. *For all $\bar{\sigma}$, if $\bar{I}(\bar{\sigma})$ then $Coh(\bar{\sigma}, \mathbb{PA} \setminus NC)$.*

Finally, it has to be shown that non-privileged cache operations preserve the equivalence of coherent and incoherent resources.

Lemma D.7.1. *Given a set of potentially incoherent addresses $A \subset \mathcal{O} \cap \mathbb{PA}$ and cache-aware states $\bar{\sigma}_1$ and $\bar{\sigma}_2$ such that*

1. $Coh(\bar{\sigma}_1, (\mathcal{O} \cap \mathbb{PA}) \setminus A)$ and $Coh(\bar{\sigma}_2, (\mathcal{O} \cap \mathbb{PA}) \setminus A)$,
2. $\forall pa \in (\mathcal{O} \cap \mathbb{PA}) \setminus A. Cv(\bar{\sigma}_1, pa) = Cv(\bar{\sigma}_2, pa)$, and
3. $(\bar{\sigma}_1.cache, \bar{\sigma}_1.mem) \approx_A (\bar{\sigma}_2.cache, \bar{\sigma}_2.mem)$,

if $\bar{\sigma}_1 \rightarrow_U \bar{\sigma}'_1 [dop]$ and $\bar{\sigma}_2 \rightarrow_U \bar{\sigma}'_2 [dop]$, and dop is cacheable, then

1. $Coh(\bar{\sigma}'_1, (\mathcal{O} \cap \mathbb{PA}) \setminus A)$ and $Coh(\bar{\sigma}'_2, (\mathcal{O} \cap \mathbb{PA}) \setminus A)$,
2. $\forall pa \in (\mathcal{O} \cap \mathbb{PA}) \setminus A. Cv(\bar{\sigma}'_1, pa) = Cv(\bar{\sigma}'_2, pa)$, and
3. $(\bar{\sigma}'_1.cache, \bar{\sigma}'_1.mem) \approx_A (\bar{\sigma}'_2.cache, \bar{\sigma}'_2.mem)$.

This lemma captures three essential arguments about the underlying hardware: (1) on coherent memory, observational equivalence is preserved w.r.t. the core-view and thus independent of whether the data is currently cached, (2) cacheable accesses cannot break the coherency of these resources, and (3) on potentially incoherent memory (addresses A), observational equivalence is preserved if cache and memory are equivalent for the corresponding lines, i.e., they have the same tag states, contents, and filtered history.

These properties inherently depend on the specific cache architecture, in particular on the eviction policy and its history filter. For any two equal filtered histories the eviction policy selects the same entry to evict (Assumption D.7.1), therefore corresponding cache states stay observationally equivalent. Moreover, they rely on the verification conditions that evictions do not change the core-view of a coherent memory resource and that cache line fills for coherent addresses read the same values from main memory.

We have formally verified Lemma D.7.1 for an instantiation of our cache model that uses an LRU replacement policy (see Appendix).

Based on the proof obligations lined out above we can now prove confidentiality for steps of the application

Lemma D.7.2. *For all $\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}'_1$ where $\bar{I}(\bar{\sigma}_1)$ and $\bar{I}(\bar{\sigma}_2)$ hold, if $\bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2$ and $\bar{\sigma}_1 \rightarrow_U \bar{\sigma}'_1$, then $\exists \bar{\sigma}'_2. \bar{\sigma}_2 \rightarrow_U \bar{\sigma}'_2$ and $\bar{\sigma}'_1 \sim_{\mathcal{O}} \bar{\sigma}'_2$.*

Proof Lemma D.7.2. We perform a case split over all possible hardware steps in non-privileged mode. Observational equivalence on cache, memory, and program counterprovide that the same instruction is fetched and executed in both steps.⁷ For hardware transitions that do not access memory, we conclude as in the proof on the cacheless model (Obligation D.7.6, similar theorems were proved in [187]).

In case of memory instructions, since general purpose registers are equivalent, the same addresses are computed for the operation. Obligations D.7.2 and D.7.3 yield that the same access permissions are in place in $\bar{\sigma}_1$ and $\bar{\sigma}_2$ such that the application can only directly read observationally equivalent memory resources. Then we distinguish cacheable and uncacheable accesses to an address pa .

In the latter case, by Obligation D.7.4 and we know that $pa \in NC$, and we obtain $\bar{\sigma}_1.mem(pa) = \bar{\sigma}_2.mem(pa)$ from $\bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2$. Now, since the access bypasses the caches, they are unchanged. Moreover, the memory accesses in both states yield the same result and the equivalence of cache and memory follows trivially.

In case of cacheable accesses we know by Obligation D.7.5 that resources $(\mathcal{O} \cap \mathbb{PA}) \setminus NC$ are coherent and we apply Lemma D.7.1 to deduce the observational equivalence of cache and memory for addresses in $(\mathcal{O} \cap \mathbb{PA}) \setminus NC$ and NC . As also register resources are updated with the same values, we conclude $\bar{\sigma}'_1 \sim_{\mathcal{O}} \bar{\sigma}'_2$. \square

⁷We do not model instruction caches in the scope of this work, but assume a unified data and instruction cache. In fact, instruction caches allow further storage-channel-based attacks [100] that would need to be addressed at this point in the proof.

D.7.4 Confidentiality: Kernel Level

Kernel level confidentiality ensures that the execution of kernel steps does not leak confidential information to the application. Specifically, this entails showing that at the end of the kernel execution observational equivalence of resources in \mathcal{O} is (re)established. First, the kernel should not leak confidential resources in absence of caches:

Proof Obligation D.7.6. *For all $\sigma_1, \sigma_2, \sigma'_1$ such that $I(\sigma_1), I(\sigma_2)$ ex-entry(σ_1), ex-entry(σ_2), and $\sigma_1 \sim_{\mathcal{O}} \sigma_2$ if $\sigma_1 \rightsquigarrow \sigma'_1$ then $\exists \sigma'_2 . \sigma_2 \rightsquigarrow \sigma'_2$ and $\sigma'_1 \sim_{\mathcal{O}} \sigma'_2$.*

The goal is now to show that the chosen countermeasure against information leakage through the caches allows transferring the confidentiality property to the cache-aware model. Formally, this countermeasure is represented by a two-state property $CM(\sigma, \sigma')$ on the cacheless and $\overline{CM}(\bar{\sigma}, \bar{\sigma}')$ on the cache-aware model. Here the first argument is the starting state of the kernel execution, while the second argument is some arbitrary state that is reached from there by a privileged computation. Property $CM(\sigma, \sigma')$ should only cover functional properties of the countermeasure that can be verified on the cacheless model as part of the kernel's internal invariant.

Proof Obligation D.7.7. *For all σ, σ' with $\sigma \rightarrow_P^* \sigma'$ and ex-entry(σ), if $II(\sigma, \sigma')$ then $CM(\sigma, \sigma')$*

The countermeasure property on the cache-aware model, on the other hand, extends CM with conditions on the cache state that prevent information leakage to the application. We demand that it can be established through the bisimulation between cacheless and cache-aware model for a given countermeasure.

Proof Obligation D.7.8. *For all $\sigma, \sigma', \bar{\sigma}, \bar{\sigma}'$ where $\sigma \rightarrow_P^* \sigma', \bar{\sigma} \rightarrow_P^* \bar{\sigma}'$, ex-entry(σ), $\bar{\sigma} \mathcal{R}_{sim} \sigma$, and $\bar{\sigma}' \mathcal{R}_{sim} \sigma'$, if $II(\sigma, \sigma')$ then $\overline{CM}(\bar{\sigma}, \bar{\sigma}')$*

Hereafter, to enable transferring non-interference properties from the cacheless model to the cache-aware model we assume that the transition relations \rightarrow_P are total functions for both models. As we want to reuse Lemma D.6.7 in the following proofs, we require a number of properties of the simulation relation and the invariants.

Lemma D.7.3. *For all $\bar{\sigma}_1, \bar{\sigma}_2, \sigma_1, \sigma_2$, such that $\bar{I}(\bar{\sigma}_1), \bar{I}(\bar{\sigma}_2), \bar{I}(\sigma_1)$, and $\bar{I}(\sigma_2)$ as well as $\bar{\sigma}_1 \mathcal{R}_{sim} \sigma_1$ and $\bar{\sigma}_2 \mathcal{R}_{sim} \sigma_2$:*

$$(1) \quad \bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2 \Rightarrow \sigma_1 \sim_{\mathcal{O}} \sigma_2 \quad (2) \quad \bar{\sigma}_1 \sim_{coh} \bar{\sigma}_2 \Leftrightarrow \sigma_1 \sim_{coh} \sigma_2$$

The properties follow directly from the definition of \mathcal{R}_{sim} , the coherency of resources in $\mathcal{O} \setminus NC$ (Obligation D.7.5), and Lemma D.6.6.

In addition we require the following technical condition on the relation of the cacheless and cache-aware invariant.

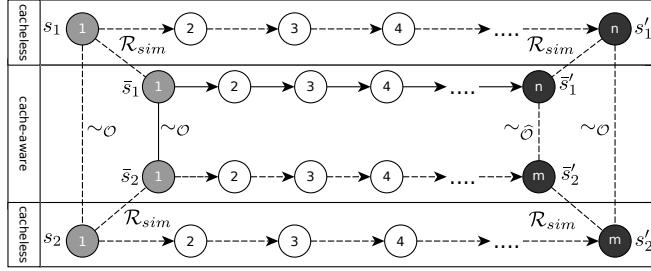


Figure D.4: Transferring confidentiality property. Dashed lines indicate proof obligations.

Proof Obligation D.7.9. *For each cache-aware state $\bar{\sigma}$ with $\bar{I}(\bar{\sigma})$, there exists a cacheless state σ such that $\bar{\sigma} \mathcal{R}_{sim} \sigma$ and $I(\sigma)$.*

Now we can use Lemma D.6.7 to transfer the equivalence of coherent resources after complete kernel executions.

Lemma D.7.4. *For all $\bar{\sigma}_1, \bar{\sigma}'_1, \bar{\sigma}_2$ such that $\bar{I}(\bar{\sigma}_1), \bar{I}(\bar{\sigma}_2)$, $\text{ex-entry}(\bar{\sigma}_1)$, $\text{ex-entry}(\bar{\sigma}_2)$, and $\bar{\sigma}_1 \rightsquigarrow \bar{\sigma}'_1$, if $\bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2$ then $\exists \bar{\sigma}'_2$. $\bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2$ and $\bar{\sigma}'_1 \sim_{coh} \bar{\sigma}'_2$.*

Proof. We follow the classical approach of mapping both initial cache-aware states to simulating ideal states by Obligation D.7.9, transferring observational equivalence by Lemma D.7.3.1, and using Lemma D.6.7 to obtain a simulating computation $\sigma_1 \rightsquigarrow \sigma'_1$ for $\bar{\sigma}_1 \rightsquigarrow \bar{\sigma}'_1$ on the cacheless level (cf. Fig. D.4). Obligation D.7.6 yields an observationally equivalent cacheless computation $\sigma_2 \rightsquigarrow \sigma'_2$ starting in the state simulating $\bar{\sigma}_2$. Applying Lemma D.6.7 again results in a corresponding cache-aware computation $\bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2$. We transfer equivalence of the coherent resources in the final states using Corollary 4 and Lemma D.7.3.2. \square

It remains to be shown that the kernel execution also maintains observational equivalence for the potentially incoherent memory resources. This property depends on the specific countermeasure used, but can be proven once and for all for a given hardware platform and cache architecture.

Proof Obligation D.7.10. *For all $\bar{\sigma}_1, \bar{\sigma}'_1, \bar{\sigma}_2$, and $\bar{\sigma}'_2$ with $\bar{\sigma}_1 \rightsquigarrow \bar{\sigma}'_1$ and $\bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2$ as well as $\bar{I}(\bar{\sigma}_1), \bar{I}(\bar{\sigma}_2)$, $\text{ex-entry}(\bar{\sigma}_1)$, $\text{ex-entry}(\bar{\sigma}_2)$, $\overline{CM}(\bar{\sigma}_1, \bar{\sigma}'_1)$ and $\overline{CM}(\bar{\sigma}_2, \bar{\sigma}'_2)$, if $\bar{\sigma}_1 \sim_{\mathcal{O}} \bar{\sigma}_2$ then*

$$(\bar{\sigma}'_1.\text{cache}, \bar{\sigma}'_1.\text{mem}) \approx_{NC} (\bar{\sigma}'_2.\text{cache}, \bar{\sigma}'_2.\text{mem}).$$

Finally, we combine above results to prove that the confidentiality of the system is preserved on the cache-aware level.

Proof Theorem D.5.3. Performing an induction on the computation length, we distinguish application and complete kernel steps. In the first case we apply Lemma D.7.2. The invariants on both states are preserved by Lemmas D.6.1,

D.6.2.1, and D.6.5. For kernel computations, we first step into non-privileged states $\bar{\sigma}_1'', \bar{\sigma}_2''$ deducing $\bar{I}(\cdot)$ and $ex\text{-}entry(\cdot)$ for both states by Theorem D.5.1, as well as $\bar{\sigma}_1'' \sim_{\mathcal{O}} \bar{\sigma}_2''$ by Lemma D.7.2. By Lemma D.7.4 we show the existence of a cache-aware computation $\bar{\sigma}_2'' \rightsquigarrow \bar{\sigma}_2'$ that preserves the equivalence of coherent resources. Using Lemma D.6.7 we obtain a corresponding cacheless computation. By Obligation D.7.8 we get $\overline{CM}(\bar{\sigma}_1'', \bar{\sigma}_1')$ and $\overline{CM}(\bar{\sigma}_2'', \bar{\sigma}_2')$. Invariants $\bar{I}(\bar{\sigma}_1')$ and $\bar{I}(\bar{\sigma}_2')$ hold due to Obligation D.6.6 and Lemma D.6.8. We conclude by Obligation D.7.10. \square

The core requirements that enable this result are Proof Obligations D.7.7, D.7.8, and D.7.10. Below we discuss how to discharge them for two common countermeasures.

D.7.5 Correctness of Countermeasures

For a given cache leakage countermeasure, one needs to define predicates CM , \overline{CM} and discharge the related proof obligations. Here we describe the countermeasures of secret-independent code and cache flushing that are well-known to work against cache timing channels. However, in the presence of uncacheable aliases they are not sufficient to prevent information leakage. In particular, if a secret-dependent value in memory is overshadowed by secret-independent dirty cache entry for the same address, the application can still extract the memory value by an uncacheable read.

To tackle this issue we require that all countermeasures sanitise writes of the kernel to addresses in NC by a cache cleaning operation. Then if the core-views of NC are equivalent in two related states, so are the memories. A sufficient verification condition on the cacheless model is imposed as follows.

Proof Obligation D.7.11. *For all σ, σ' such that $CM(\sigma, \sigma')$ and $\sigma \rightsquigarrow \sigma'$ performs $dop_1 \cdots dop_n$ with $dop_i = wt(R_i)$ and $pa \in R_i$, if $pa \in NC$ then there is a $j > i$ such that $dop_j = cl(R_j)$ and $pa \in R_j$.*

This condition is augmented with countermeasure-specific proof obligations that guarantee the observational equivalence of caches.

Secret-Independent Code

The idea behind this countermeasure is that no information can leak through the data caches, if the kernel accesses the same memory addresses during all computations from observationally equivalent entry states, i.e., the memory accesses do not depend on the confidential resources. We approximate this functional requirement by predicate $CM(\sigma, \sigma')$:

Proof Obligation D.7.12. *If a complete kernel execution $\sigma \rightsquigarrow \sigma'$ performs $dop_1 \cdots dop_n$ on resources $R_1 \cdots R_n$, then for all σ_2, σ'_2 with $\sigma \sim_{\mathcal{O}} \sigma_2$ and $\sigma_2 \rightsquigarrow \sigma'_2$ operations $dop'_1 \cdots dop'_n$ on $R'_1 \cdots R'_n$ are performed such that addresses in R_i and R'_i map to the same set of tags.*

Note that this allows to access different addresses depending on a secret, as long as both addresses have the same tag. Such trace properties can be discharged by a relational analysis at the binary level [24]. On the cache-aware level, the property is refined by:

$$\overline{CM}(\bar{\sigma}, \bar{\sigma}') \stackrel{\text{def}}{=} \forall \bar{\sigma}_2, \bar{\sigma}'_2. \bar{\sigma} \rightsquigarrow \bar{\sigma}' \wedge \bar{\sigma} \approx_{NC} \bar{\sigma}_2 \wedge \bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2 \Rightarrow \bar{\sigma}' \approx_{NC} \bar{\sigma}'_2$$

The refinement (Obligation D.7.8) is proven similar to Lemma D.7.4 using the Lemma D.6.7 and the determinism of the hardware when applying the confidentiality of the system in the cacheless model (Lemma D.7.6) as well as Obligation D.7.11. Then, proving Obligation D.7.10 is straightforward.

Cache Flushing

A simple way for the kernel to conceal its secret-dependent operations is to flush the cache before returning to the application. The functional requirement $CM(\sigma, \sigma')$ implies:

Proof Obligation D.7.13. *For any kernel computation $\sigma \rightsquigarrow \sigma'$ performing data operations $dop_1 \dots dop_n$:*

1. *there exists a contiguous subsequence $dop_i \dots dop_j$ of clean operations on all cache lines,*
2. *operations $dop_1 \dots dop_j$ do not write resources in NC,*
3. *operations $dop_{j+1} \dots dop_n$ accessed address tags and written values for NC do not depend on confidential resources,*

Condition (3) is formalised and proven like the secret-independent code countermeasure discussed above. Condition (1) can be verified by binary code analysis, checking that the expected sequence of clean operations is eventually executed. We identify the resulting state by $fl(\sigma)$. Condition (2) is not strictly necessary, but it reduces the overall verification effort. Then we demand by $\overline{CM}(\bar{\sigma}, \bar{\sigma}')$:

Definition. *For all $\bar{\sigma}''$ such that $\bar{\sigma} \rightarrow_P^* \bar{\sigma}'' \rightarrow_P^* \bar{\sigma}'$:*

1. *if $fl(\bar{\sigma}'')$ or $\bar{\sigma}''$ is a preceding state in the computation, then for all $pa \in NC$ it holds that $Mv(\bar{\sigma}'', pa) = Mv(\bar{\sigma}, pa)$,*
2. *if $fl(\bar{\sigma}'')$, all cache lines and their filtered histories are empty,*
3. *if $fl(\bar{\sigma}'')$ and $\bar{\sigma}'' \rightsquigarrow \bar{\sigma}'$, then for all computations $\bar{\sigma}_2 \rightsquigarrow \bar{\sigma}'_2$ with $\bar{\sigma}'' \approx_{NC} \bar{\sigma}_2$ we have $\bar{\sigma}' \approx_{NC} \bar{\sigma}'_2$.*

Here, Condition (1) holds as resources NC are not written and are affected only by cache evictions which preserve the memory-view. Condition (2) follow directly from the cache-flush semantics. Condition (3) is discharged using Lemma D.6.7 and Obligation D.7.13.3. In the proof of Obligation D.7.10 we establish memory equivalence between intermediate states $\bar{\sigma}_1''$ and $\bar{\sigma}_2''$ where $fl(\bar{\sigma}_1'')$ and $\bar{\sigma}_2''$ using Condition (1). By Condition (2) we obtain $\bar{\sigma}_1'' \approx_{NC} \bar{\sigma}_2''$ and conclude by Condition (3).

D.8 Case Study

As a case study we use a real hypervisor capable of hosting a Linux guest along with security services that has been formally verified previously on a cacheless model [71, 101] and vulnerable to attacks based on cache storage channel [100].

A hypervisor is a system software which controls access to resources and can be used to create isolated partitions on a shared hardware. The hypervisor paravirtualizes the platform for several guests. Only the hypervisor executes in privileged mode, while guests entirely run in non-privileged mode and need to invoke hypervisor functionalities to modify critical resources, such as page-tables. The hypervisor uses direct paging [157] to virtualize the memory subsystem. Using this approach a guest prepares a page-table in its own memory, which after validation, is used by the hypervisor to configure the MMU, without requiring memory copy operations. For validated page-tables the hypervisor ensures that the guest has no writable accesses to the page-tables, thus ensuring that the guest cannot change the MMU configuration. This mechanism makes the hypervisor particularly relevant for our purpose since the critical resources change dynamically and ownership transfer is used for communication.

To efficiently implement direct paging, the hypervisor keeps a type (either *page-table* or *data*) and a reference counter for each physical memory page. The counter tracks (i) for a data-page the number of virtual aliases that enable non-privileged writable accesses to this page, and (ii) for a page-table the number of times the page is used as page-table. The intuition is that the hypervisor can change type of a page (e.g., when it allocates or frees a page-table) only if the corresponding reference counter is zero.

The security of this system can be subverted if page-tables are accessed using virtual aliases with mismatched cacheability attributes: The application can potentially mislead the hypervisor to validate stale data and to make a non-validated page a page-table. The hypervisor must counter this threat by preventing incoherent memory from being a page-table. Here, we fix the vulnerability forcing the guest to create page-tables only inside an always cacheable region of the memory.

We instantiate the general concepts of Section D.5.1 for the case study hypervisor. The hypervisor uses a static region of physical memory HM to store its stack, data structures and code. This region includes the data structure used to keep the reference counter and type for memory pages. Let $T(\sigma, pa) = pt$ represent that the

hypervisor data-structure types the page containing the address pa as *page-table*, then the critical resources are $CR(\sigma) = HM \cup \{pa. T(\sigma, pa) = pt\}$.

The state invariant \bar{I} guarantees: (i) soundness of the reference counter, (ii) that the state of the system is well typed; i.e., the MMU uses only memory pages that are typed *page-table*, and page-tables forbid non-privileged accesses to pages outside HM or not typed data. Since the hypervisor uses always cacheability, the invariant also requires (iii) that $HM \subseteq M_{ac}$, if $T(\sigma, pa) = pt$ then $pa \in M_{ac}$, coherency of M_{ac} , and that all aliases to M_{ac} are cacheable.

Obligation D.6.1.1 is demonstrated by showing that the functional part of the invariant only depends on page-tables and the internal data-structures, which are critical and contained in HM . Obligations D.6.1.2 and D.6.1.3 are demonstrated by correctness of always cacheability. Obligation D.6.2 trivially holds, since type of memory pages only depends on the internal data structure of the hypervisor, which is always considered critical. Property (ii) of the invariant guarantees Obligation D.6.3, since the MMU can use only memory blocks that are typed page-table, which are considered critical. The same property ensures that all critical resources are not writable by the application, thus guaranteeing Obligation D.4. Moreover, Obligations D.6.5 and 1 are guaranteed by soundness of the countermeasure. Obligation D.6.8 is proved by showing that the functional invariants for the two models are defined analogously using the core-view. Finally, property (iii) guarantees countermeasure specific Obligations D.6.11.1 and D.6.11.2.a.

There remains to verify obligations on the hypervisor code. This can be done using a binary analysis tool, since the obligations are defined solely on the cache-less model: (Obligation D.6.6) the hypervisor handlers are correct, preserving the functional invariant, (Obligation D.6.7) the control flow graph of the hypervisor is correct and the hypervisor never changes its own memory mapping, (Obligation D.6.11.2.b) all memory accesses of the hypervisor are in the always cacheable region.

D.9 Implementation

We used the HOL4 interactive theorem prover [114] to validate our security analysis. We focused the validation on Theorems D.5.1 and D.5.2, since the integrity threats posed by storage channels cannot be countered by means external to model (e.g., information leakage can be neutralised by introducing noise in the cache state).

Following the proof strategy of Section D.6, the proof has been divided in three layers: an abstract verification establishing lemmas that are platform- and countermeasure-independent, the verification of soundness of countermeasures, and a part that is platform-specific. For the latter, we instantiated the models for both ARMv7 and ARMv8, using extensions of the models of Anthony Fox [82, 2] that include the cache model of Section D.11.1. The formal specification used in our analysis consists of roughly 2500 LOC of HOL4, and the complete proof consists of 10000 LOC.

For the case study we augmented the existing hypervisor [71, 101] with the always cacheability countermeasure. This entailed some engineering effort to adapt the memory allocator of the Linux kernel to allocate page-tables inside M_{ac} . The adaptation required changes to 45 LoC in the hypervisor and an addition of 35 LoC in the paravirtualized Linux kernel and imposes a negligible performance overhead. The formal model of the hypervisor has been modified to include the additional checks performed by the hypervisor to prevent allocation of page-tables outside M_{ac} and to forbid uncacheable aliases to M_{ac} . Similarly, we extended the functional invariant with the new properties guaranteed by the adopted countermeasure. The model of the hypervisor has been used to show that the new design preserves the functional invariant. The invariant and the formal model of ARMv7 processor have been used to validate the proof obligations that do not require binary code analysis.

We did not analyse the binary code of the hypervisor. However, we believe that the proof of the corresponding proof obligations can be automated to a large extent using binary analysis tools (e.g. [101]) or using refinement techniques (e.g. [190]).

Finally, we validated the analysis of Section D.7.3, instantiating the verification strategy for the formal model of ARMv7 processor and the flushing countermeasure.

D.10 Conclusion

We presented an approach to verify countermeasures for cache storage channels. We identified the conditions that must be met by a security mechanism to neutralise the attack vector and we verified correctness of some of the existing techniques to counter both integrity and confidentiality attacks.

The countermeasures are formally modelled as new proof obligations that can be imposed on the cacheless model to ensure the absence of vulnerability due to cache storage channels. The result of this analysis are theorems in Section D.5.3. They demonstrate that a software satisfying a set of proof obligations (i.e., correctly implementing the countermeasure) is not vulnerable because of cache storage channels. Since these proof obligations can be verified using a memory coherent setting, existing verification tools can be used to analyse the target software. For example, the proof obligations required to demonstrate that a countermeasure is in place (e.g. D.6.11, D.6.12, D.7.11, D.7.12) can be verified using existing binary analysis tools.

While this paper exemplifies the approach for unified single-level data-caches, our methodology can be extended to counter leakage through timing channels and accommodate more complex scenarios and other hardware features too. For instance our approach can be used to counter storage channels due to enabling multi-core processing, multi-level caches, instruction caches, and TLB.

In a multi-core setting the integrity proof can be straightforwardly adopted. However, for confidentiality new countermeasures such as stealth memory or cache partitioning must be used to ensure that secret values cannot be leaked. This entails defining new proof obligations to make sure that the countermeasures are correctly

implemented and protect secret values. In the STEALTHMEM approach [129] each core is given exclusive access to a small portion of the shared cache for its security critical computations. By ensuring that this stealth memory is always allocated in the cache, and thus no eviction can happen, one can guarantee that no storage channel can be built.

Multi-level caches can be handled iteratively in a straightforward fashion, starting from the cacheless model and adding CPU-closer levels of cache at each iteration. Iterative refinement has three benefits: Enabling the use of existing (cache unaware) analysis tools for verification, enabling transfer of results of sections D.6.2, D.6.3, D.7.4 and D.7.5 to the more complex models, and allowing to focus on each hardware feature independently, so at least partially counteracting the pressure towards ever larger and more complex global models. For non-privileged transitions the key tools are derivability and Lemmas D.6.2 and D.7.1. These fit new hardware features well. For instance, for separate instruction- and data-caches, coherency should be extended to require that instruction-cache hits are equal to core-view (and when this is not the case the address should then become attacker observable). Since derivability is not instruction dependent, Lemma D.6.2 (and consequently Theorem D.5.1) can be easily re-verified.

It worth noting that sometimes it is possible to transfer by refinement also properties for non-privileged transitions. This is the case for TLBs without virtualisation extensions: As non-privileged instructions are unable to directly modify the TLB, incoherent behaviours can arise only by the assistance of kernel transitions. If a refinement has been proved for kernel transitions (i.e., that the TLB is properly handled by the kernel), then a refinement can be established for the non-privileged transitions too.

In general, the verification of refinement for privileged transitions requires to show (i) that the refinement is established if a countermeasure is in place, (ii) that the countermeasure ensures the kernel to not leave secret dependent footprints in incoherent resources, and (iii) that the kernel code implements the countermeasures. For instance, instruction-caches require that instructions fetched by the kernel is the same in both models, e.g., because the kernel code is not self-modifying or because the caches are flushed before executing modified code. For confidentiality, *pc*-security can be shown to guarantee that instruction-cache is not affected by secrets.

Note also that the security analysis requires trustworthy models of hardware, which are needed to verify platform-dependent proof obligations. Some of these properties (e.g., Assumption D.7.1) require extensive tests to demonstrate that corner cases are correctly handled by models. For example, while the conventional wisdom is that flushing caches can close side-channels, a new study [88] showed flushing does not sanitise caches thoroughly and leaves some channels active, e.g. instruction cache attack vectors.

There are several open questions concerning side-channels due to similar shared low-level hardware features such as branch prediction units, which undermine the soundness of formal verification. This is an unsatisfactory situation since formal

proofs are costly and should pay off by giving reliable guarantees. Moreover, the complexity of contemporary hardware is such that a verification approach allowing reuse of models and proofs as new hardware features are added is essential for formal verification in this space to be economically sustainable. Our results represent a first step towards giving these guarantees in the presence of low level storage channels.

$$\begin{aligned}
SL(C, i) &\equiv C(i).slice, \quad W(C, i, t) \equiv (SL(C, i))(t), \quad H(C, i) \equiv C(i).hist, \\
c\text{-}cnt(C, pa) &\equiv W(C, i, t).D(wi), \quad c\text{-}hit(C, pa) \equiv W(C, i, t) \neq \perp, \\
c\text{-}dirty(C, pa) &\equiv c\text{-}hit(C, pa) \wedge W(C, i, t).d
\end{aligned}$$

Figure D.5: Abbreviations; for $pa \in \mathbb{PA}$ mapped by $va \in \mathbb{VA}$ and where i , t and wi are the corresponding set index, tag and word index, and $C \in \mathbb{C}$.

D.11 Appendix

D.11.1 Generic Data Cache Model

Below we give the definition of our generic cache model which is the basis for discharging Assumptions D.7.1, D.7.1 and Proof Obligations D.7.8, D.7.10.

Our cache model does not fix the cache size, the number of ways, the format of set indices, lines, tags, or the eviction policy. Moreover, cache lines can be indexed according to physical or virtual addresses, but are physically tagged, and our model allows both write-back and write-through caching.

Let n and m be the bitsize of virtual and physical addresses, then $\mathbb{VA} = \mathbb{B}^{n-\alpha}$ and $\mathbb{PA} = \mathbb{B}^{m-\alpha}$ are the sets of word-aligned addresses where 2^α is the word size in number of bytes (e.g. in a 64-bit architecture $\alpha = 3$). Our cache has 2^N sets, its lines are of size 2^L words and we use functions $si : \mathbb{VA} \times \mathbb{PA} \rightarrow \mathbb{B}^T$, $tag : \mathbb{PA} \rightarrow \mathbb{B}^T$, and $widx : \mathbb{VA} \times \mathbb{PA} \rightarrow \mathbb{B}^L$ to compute the set indices, tags and word indices of the cache. We have $T = m - (N + L)$ for physically-indexed and $T = m - L$ for virtually-indexed caches.

We define a *cache slice* as a mapping from a tag to a cache line, $\mathbb{SL} = \mathbb{B}^T \rightarrow \mathbb{L} \cup \{\perp\}$; \perp if no line is cached for the given tag. A line $ln \in \mathbb{L}$ is a pair $(ln.D, ln.d) \in \mathbb{D} \times \mathbb{B}$, \mathbb{D} is the mapping from word-indices to data, and d indicates the line dirtiness.

Then, a cache $C \in \mathbb{C}$ maps a set index i to a slice $C(i).slice \in \mathbb{SL}$ and a history of actions performed $C(i).hist \in \mathbb{A}^*$ on that slice, i.e., $\mathbb{C} = \mathbb{B}^N \rightarrow \mathbb{SL} \times \mathbb{A}^*$. The history records internal cache actions of type $\mathbb{A} = \{\text{touch}_{r|w} t, \text{evict } t, \text{lfill } t\}$, where $t \in \mathbb{B}^T$ denotes the tag associated with the action.

Here, $\text{touch}_{r|w} t$ denotes a read or write access to a line tagged with t , $\text{lfill } t$ occurs when a line for tag t is loaded from memory and placed in the cache. Similarly, $\text{evict } t$ represents the eviction of a line with tag t . To simplify formalization of properties, we define a number of abbreviations in Figure D.5. Note that in case of virtual indexing, we assume an unaliased cache, i.e., each physical address is cached in at most one cache slice.

The semantics of internal cache actions on a cache slice and history are given by the corresponding functions in Figure D.6. If tag t hits the cache in the line set i , then *touch* and *evict* update the cache $C \in \mathbb{C}$ as follows, where $D'_i = W(C, i, t).D[w_i \mapsto v']$ is the resulting line of a write operation at word index wi

$$\begin{aligned}
\text{touch} &: \mathbb{SL} \times \mathbb{A}^* \times \mathbb{B}^T \times \mathbb{B}^L \times (\mathbb{B}^w \cup \{\perp\}) \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
\text{lfill} &: \mathbb{SL} \times \mathbb{A}^* \times \mathbb{M} \times \mathbb{PA} \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
\text{evict} &: \mathbb{SL} \times \mathbb{A}^* \times \mathbb{B}^T \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
\text{wriba} &: \mathbb{L} \times \mathbb{M} \rightarrow \mathbb{M} \quad \text{evict?} : \mathbb{A}^* \times \mathbb{B}^T \rightarrow \mathbb{B}^T \cup \perp
\end{aligned}$$

Figure D.6: Internal operations of the cache. The fifth input of *touch* is either \perp for read accesses or the value $v' \in \mathbb{B}^w$ being written to the cache line.

with value v' :

$$(SL(C', i), H(C', i)) := \begin{cases} (SL(C, i), H(C, i) @ (\text{touch}_r t)), & : \text{touch}_r t \\ (SL(C, i)[t \mapsto (D'_i, 1)], H(C, i) @ (\text{touch}_w t)), & : \text{touch}_w t \\ (SL(C, i)[t \mapsto \perp], H(C, i) @ (\text{evict } t)), & : \text{evict } t \end{cases}$$

For cache misses on a physical address pa with tag t , function *lfill* loads memory content $v = \text{mem}[pa + L \cdot 2^\alpha - 1 : pa]$ and places it as a clean line into the cache: $(SL(C', i), H(C', i)) := (SL(C, i)[t \mapsto (\lambda i. v[i], 0)], H(C, i) @ (\text{lfill } t))$.

As the cache can only store limited amounts of lines, eviction policy *evict?* (H, t) returns the tag of the line to be replaced at a line fill for tag t , or \perp if eviction is not required. Evicted dirty lines are then written back into the memory using function *wriba*. As explained in the main text, the eviction policy is only depending on a finite subset of the history represented by filter function $\varphi : \mathbb{A}^* \rightarrow \mathbb{A}^*$ (Assumption D.7.1).

The definitions above provide the minimal building blocks to define a detailed cached memory system that responds to reads, writes, and eviction requests from the core. Below we give an example.

D.11.2 Operational Write-back Cache Semantics

With the help of functions in Fig. D.6 and abbreviations of Fig. D.5. we give semantics to a write-back cache with LRU replacement strategy. Figure D.7 lists the interface available to the core to control the cache. When one of these functionalities is called, the cache uses the internal actions to update its state according to the requested operation. In what follows we set $t = \text{tag}(va, pa)$ and $i = \text{si}(va, pa)$.

Function *fill* (C, M, va, pa) loads the cache $C \in \mathbb{C}$ by invoking *lfill*. However, if the cache is full the eviction policy determines the line to evict to make space. Using *wriba*, the evicted line is then written back in the memory $M \in \mathbb{M}$. We denote this conditional eviction by *alloc* (C, M, va, pa), which is defined as:

$$\begin{cases} (C[i \mapsto \text{evict}(SL(C, i), H(C, i), t')], \text{wriba}(W(C, i, t'), M)) : \text{evict?}(H(C, i), t) = t' \\ (C, M) : \text{evict?}(H(C, i), t) = \perp \end{cases}$$

We save the result of this function as the pair (\bar{C}, \bar{M}) . Moreover, if an alias for the filled line is present in another cache slice, i.e., a line with the same tag, that line

$$\begin{aligned}
fill &: \mathbb{C} \times \mathbb{M} \rightarrow \mathbb{V}\mathbb{A} \times \mathbb{P}\mathbb{A} \rightarrow \mathbb{C} \times \mathbb{M} \\
read &: \mathbb{C} \times \mathbb{M} \times \mathbb{V}\mathbb{A} \times \mathbb{P}\mathbb{A} \rightarrow \mathbb{C} \times \mathbb{M} \times \mathbb{B}^w \\
write &: \mathbb{C} \times \mathbb{M} \times \mathbb{V}\mathbb{A} \times \mathbb{P}\mathbb{A} \times \mathbb{B}^w \rightarrow \mathbb{C} \times \mathbb{M} \\
invba &: \mathbb{C} \times \mathbb{M} \times \mathbb{V}\mathbb{A} \times \mathbb{P}\mathbb{A} \rightarrow \mathbb{C} \times \mathbb{M} \\
clnba &: \mathbb{C} \times \mathbb{M} \times \mathbb{V}\mathbb{A} \times \mathbb{P}\mathbb{A} \rightarrow \mathbb{C} \times \mathbb{M}
\end{aligned}$$

Figure D.7: Core accessible interface. Functions *clnba* and *invba* clean and invalidate cache lines for given virtual and physical addresses; *clnba* only resets the dirty bit and writes back dirty lines, while *invba* also evicts the line. Function *fill* is used to pre-load lines into the cache.

has to be evicted as well. We define this condition as follows:

$$\text{alias?}(C, t, i, i') \equiv \exists va', pa'. i' = si(va', pa') \wedge i \neq i' \wedge W(C, si(va', pa'), t) \neq \perp$$

Then alias detection and eviction *alias*(C, M, va, pa) is defined as:

$$\begin{cases} (C[i' \mapsto \text{evict}(SL(C, i'), H(C, i'), t)], wriba(W(C, i', t), M)) : \text{alias?}(C, t, i, i') \\ (C, M) : \text{otherwise} \end{cases}$$

The result of this function applied to (\bar{C}, \bar{M}) is saved as (\hat{C}, \hat{M}) . The combination of these actions with a line fill is denoted by *fillwb*(C, M, va, pa) and defined below.

$$\begin{cases} (\hat{C}[i \mapsto lfill(SL(\hat{C}, i), H(\hat{C}, i), M, pa)], \hat{M}) : \neg c\text{-hit}(C, pa) \\ (C, M) : \text{otherwise} \end{cases}$$

Thus, $(\tilde{C}, \tilde{M}) = \text{fillwb}(C, M, pa, va)$. Now the definition of reading, writing, flushing, and cleaning the cache is straightforward, for $x = C, M, va, pa$ we have:

$$\begin{aligned}
\text{read}(x) &= (\tilde{C}[i \mapsto \text{touch}(SL(\tilde{C}, i), H(\tilde{C}, i), t, i, \perp), \tilde{M}, c\text{-cnt}(\tilde{C}, pa)]) \\
\text{write}(x, v) &= (\tilde{C}[i \mapsto \text{touch}(SL(\tilde{C}, i), H(\tilde{C}, i), t, i, v), \tilde{M}) \\
\text{invba}(x) &= (C[i \mapsto \text{evict}(SL(C, i), H(C, i), t)], wriba(W(C, i, t), M)) \\
\text{clnba}(x) &= (C[i \mapsto SL(C, i)[t \mapsto W(C, i, t)[d \mapsto 0]]], wriba(W(C, i, t), M))
\end{aligned}$$

Other cache functionalities can be defined similarly. It remains to instantiate the eviction policy and its filter φ . We choose the Least Recently Used (LRU) policy, which always replaces the element that was not used for the longest time if there is no space. In a k set associative cache we model LRU as a decision queue $q \in \mathbb{Q}$ of size k . This queue maintains an order on how the tags in a cache set are accessed. In this ordering the queue's front is the one that has been touched most recently and its back points to the tag to be replaced next upon a miss (or an empty way).

We assume two functions to manipulate queue content: 1. *push* : $\mathbb{Q} \times \mathbb{B}^T \rightarrow \mathbb{Q}$ adjusts the queue to make room for the coming tag, inserting the tag at the front of the queue, and 2. *pop* : $\mathbb{Q} \times \mathbb{B}^T \rightarrow \mathbb{Q}$ removes the input tag and shifts all elements

to the front. Additionally $back : \mathbb{Q} \rightarrow \mathbb{B}^T \cup \perp$ returns the back ($k - 1$ th) element in the queue or \perp if there is still space. We construct the queue recursively from the history h with the function $Cons : h \rightarrow \mathbb{Q}$:

$$\begin{aligned} Cons(\varepsilon) &= \emptyset \\ Cons(h@a) &= \begin{cases} pop_t(Cons(h)) & : a = \text{evict } t \\ push_t(Cons(h)) & : a = \text{lfill } t \\ push_t(pop_t(Cons(h))) & : a = \text{touch}_{r|w} t \end{cases} \end{aligned}$$

Then the eviction policy evict? is defined by:

$$\text{evict?}_{lru}(h, t) := back(Cons(h))$$

Proposition 1. *On each cache slice, the LRU replacement strategy depends only on the action history for the (at most k) present tags, since their lines were last filled into the cache.*

We capture this part of the action history through the filter function $\varphi_{lru} : \mathbb{A}^* \rightarrow \mathbb{A}^*$. Its definition depends on a number of helper functions. First we introduce the last action $last(h, t)$ on a tag t in history h and the set of tags $T_h \subset \mathbb{B}^T$ that are currently present in a slice according to history information. For action $a \in \mathbb{A}$, $tag(a)$ returns the associated tag.

$$\begin{aligned} last(\varepsilon, t) &= \perp & last(h@a, t) &= \begin{cases} a & : tag(a) = t \\ last(h, t) & : \text{otherwise} \end{cases} \\ T_h &= \{t \mid last(h, t) \notin \{\perp, \text{evict } t\}\} \end{aligned}$$

For a set of tags $T \subset \mathbb{B}^T$ we define the LRU filter includes the least recent fills and subsequent touches to tags in the set, but leaves out evictions and actions on irrelevant tags.

$$\begin{aligned} \varphi_{lru}(\varepsilon, T) &= \varepsilon & \varphi_{lru}(h, \emptyset) &= \varepsilon \\ \varphi_{lru}(h@a, T) &= \begin{cases} \varphi_{lru}(h, T)@a & : \exists t. a = \text{touch}_{r|w} t \wedge t \in T \\ \varphi_{lru}(h, T \setminus \{t\})@a & : \exists t. a = \text{lfill } t \wedge t \in T \\ \varphi_{lru}(h, T) & : \text{otherwise} \end{cases} \end{aligned}$$

Then we set $\varphi_{lru}(h) := \varphi_{lru}(h, T_h)$.

D.11.3 Proof of Assumption D.7.1

Proving the claim

$$\text{evict?}(h, t) = \text{evict?}(\varphi_{lru}(h), t)$$

boils down to the following property.

Lemma D.11.1. *For all $h \in \mathbb{A}^*$, $\text{Cons}(h) = \text{Cons}(\varphi_{lru}(h))$.*

It is proven by induction on the length of h , with one additional induction hypothesis:

$$\forall t \in T_h. \text{pop}_t(\text{Cons}(h), t) = \text{Cons}(\varphi_{lru}(h, T_h \setminus \{t\}))$$

We need two invariants of the cache semantics for histories $h' = h@a$: 1. if a is an eviction or a touch on tag t , then $t \in T_h$, and 2. if a is a line fill of tag t then $t \notin T_h$. Moreover, we use idempotence and commutativity properties of applying $\varphi_{lru}(h, T)$ several times.

$$\begin{aligned} \varphi_{lru}(h, T) &= \varphi_{lru}(\varphi_{lru}(h, T), T) \\ \varphi_{lru}(\varphi_{lru}(h, T), T') &= \varphi_{lru}(\varphi_{lru}(h, T'), T) \end{aligned}$$

With these arguments the detailed proof becomes a straightforward exercise. We also introduce the following lemmas for later use.

Lemma D.11.2. *For two histories h_1, h_2 and two tag sets T and $T' \subseteq T$, if $\varphi_{lru}(h_1, T) = \varphi_{lru}(h_2, T)$ then $\varphi_{lru}(h_1, T') = \varphi_{lru}(h_2, T')$.*

We prove this property by induction on the length of the filtered histories using the definition of $\varphi_{lru}(h, T)$.

Lemma D.11.3. *All cache actions preserve the following invariant on slices i of cache C :*

$$T_{H(C,i)} = \{t \mid W(C, i, t) \neq \perp\}$$

This follows directly from the semantics of the cache actions.

D.11.4 Proof of Assumption D.7.1

Before we conduct the proof we first give a formal definition of observational equivalence of cache and memory wrt. a set of addresses A . Let $TG_A = \{\text{tag}(a) \mid a \in A\}$ and $SI_A = \{si(a) \mid a \in A\}$ denote the sets of tags and set indices corresponding to addresses in A , then:

$$\begin{aligned} (C_1, M_1) \approx_A (C_2, M_2) &\stackrel{\text{def}}{=} \\ \forall pa \in A. M_1(pa) &= M_2(pa) \\ \wedge \forall i, t. W(C_1, i, t) &= \perp \Leftrightarrow W(C_2, i, t) = \perp \\ \wedge \forall i, t. W(C_1, i, t).d &= W(C_2, i, t).d \\ \wedge \forall i \in SI_A, t \in TG_A. & \\ W(C_1, i, t) \neq \perp &\Rightarrow W(C_1, i, t) = W(C_2, i, t) \\ \wedge \forall i. \varphi(H(C_1, i)) &= \varphi(H(C_2, i)) \end{aligned}$$

Now let $C_1 = \bar{\sigma}_1.\text{cache}$, $C_2 = \bar{\sigma}_2.\text{cache}$, $M_1 = \bar{\sigma}_1.\text{mem}$, $M_2 = \bar{\sigma}_2.\text{mem}$, and similar for the primed states, then it needs to be shown:

$$\begin{aligned}
& \forall \bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}'_1, \bar{\sigma}'_2, dop. \\
& \quad \text{Coh}(\bar{\sigma}_1, (\mathcal{O} \cap \mathbb{PA}) \setminus A) \\
& \quad \wedge \text{Coh}(\bar{\sigma}_2, (\mathcal{O} \cap \mathbb{PA}) \setminus A) \\
& \quad \wedge \forall pa \in (\mathcal{O} \cap \mathbb{PA}) \setminus A. Cv(\bar{\sigma}_1, pa) = Cv(\bar{\sigma}_2, pa) \\
& \quad \wedge (C_1, M_1) \approx_A (C_2, M_2) \\
& \quad \wedge \bar{\sigma}_1 \rightarrow_U \bar{\sigma}'_1 [dop] \\
& \quad \wedge \bar{\sigma}_2 \rightarrow_U \bar{\sigma}'_2 [dop] \\
\implies & \quad \text{Coh}(\bar{\sigma}'_1, (\mathcal{O} \cap \mathbb{PA}) \setminus A) \\
& \quad \wedge \text{Coh}(\bar{\sigma}'_2, (\mathcal{O} \cap \mathbb{PA}) \setminus A) \\
& \quad \wedge \forall pa \in (\mathcal{O} \cap \mathbb{PA}) \setminus A. Cv(\bar{\sigma}'_1, pa) = Cv(\bar{\sigma}'_2, pa) \\
& \quad \wedge (C'_1, M'_1) \approx_A (C'_2, M'_2)
\end{aligned}$$

In general, cache operations cannot make coherent resources incoherent, thus we can focus on the last two claims. All memory instructions are broken down into a sequence of internal cache actions so it suffices to make a case split on the possible cache actions $a \in \mathbb{A}$. We outline the general proof strategy for each case below.

$a = \text{touch}_r t$ — A read hit does not change contents of cache and memory at all.

We only need to consider the changes to the action history of the affected slice i . By definition of φ_{lru} we have:

$$\begin{aligned}
\varphi_{lru}(H(C'_1, i)) &= \varphi_{lru}(H(C_1, i) @ a) \\
&= \varphi_{lru}(H(C_1, i)) @ a \\
&= \varphi_{lru}(H(C_2, i)) @ a \\
&= \varphi_{lru}(H(C_2, i) @ a) \\
&= \varphi_{lru}(H(C'_2, i))
\end{aligned}$$

$a = \text{touch}_w t$ — The case of write hits is analogous to the read case, with the exception that the data content and dirty bit may change. Nevertheless the written line is present in both caches with the same contents and the dirty bit becomes 1 in both states after the write operation. Since the same value is written, we can also show the claim that the data content for tags $t \in TG_A$ are equal.

$a = \text{lfill } t$ — A line fill leaves the memory and dirty bits unchanged and since we have the same tag states, the line fill occurs in both caches.

For tags $t \notin TG_A$ that belong to coherent addresses we know that the core-view stays unchanged because a line is only fetched if it was not present in the cache before and the memory content that was visible in the core-view of the pre-state is loaded into the cache to be visible in the core-view of the post-state.

For $t \in TG_A$, relation \approx_A guarantees the equivalence of the memory contents directly for addresses A and again the same line is filled into the cache.

In both cases, the tag states stay equivalent because the same tag is added into the cache slice. Concerning the history of the cache slice, we get from the definition of φ_{lru} with $h_1 = H(C_1, i)$ and $h_2 = H(C_2, i)$:

$$\begin{aligned}
 \varphi_{lru}(H(C'_1, i)) &= \varphi_{lru}(h_1 @ a) \\
 &= \varphi_{lru}(h_1 @ a, T_{h_1 @ a}) \\
 &= \varphi_{lru}(h_1 @ a, T_{h_1} \cup \{a\}) \\
 &= \varphi_{lru}(h_1, T_{h_1}) @ a \\
 &= \varphi_{lru}(h_1) @ a \\
 &= \varphi_{lru}(h_2) @ a \\
 &= \varphi_{lru}(h_2, T_{h_2}) @ a \\
 &= \varphi_{lru}(h_2 @ a, T_{h_2} \cup \{a\}) \\
 &= \varphi_{lru}(h_2 @ a, T_{h_2 @ a}) \\
 &= \varphi_{lru}(h_2 @ a) \\
 &= \varphi_{lru}(H(C'_2, i))
 \end{aligned}$$

$a = \text{evict } t$ — For coherent resources evictions do not change the core-view, as any line that is evicted was either dirty before and thus written back to memory, maintaining its addresses' core-view, or it was clean but coherent with the corresponding memory content that becomes visible in the core-view after the eviction. If a confidential line is evicted there is nothing more to show.

For tags in a line i tag states and filtered histories are equal. By Assumption D.7.1 the eviction policy yields the same result in both states, thus if a line is evicted it is done so in both caches and these lines have the same tag.

For evicted coherent lines or confidential lines we argue as above. For lines belonging to the set A we know that they have the same contents, so if they are dirty, memory changes in the same way. In case they are clean, memories stay unchanged and are still equivalent.

In all cases the tag state is manipulated in the same way, as the same tags are evicted, thus they stay equal. The filtered histories for line i are still the same by definition of φ_{lru} and the equality of tag states.

$$\begin{aligned}
 \varphi_{lru}(H(C'_1, i)) &= \varphi_{lru}(h_1 @ a) \\
 &= \varphi_{lru}(h_1 @ a, T_{h_1 @ a}) \\
 &= \varphi_{lru}(h_1 @ a, T_{h_1} \setminus \{a\}) \\
 &= \varphi_{lru}(h_1, T_{h_1} \setminus \{a\}) \\
 &= \varphi_{lru}(h_1, T_{h_2} \setminus \{a\}) \quad (\text{Lemma D.11.3}) \\
 &= \varphi_{lru}(h_2, T_{h_2} \setminus \{a\}) \quad (\text{Lemma D.11.2}) \\
 &= \varphi_{lru}(h_2 @ a, T_{h_2} \setminus \{a\}) \\
 &= \varphi_{lru}(h_2 @ a, T_{h_2 @ a}) \\
 &= \varphi_{lru}(h_2 @ a) \\
 &= \varphi_{lru}(H(C'_2, i))
 \end{aligned}$$

This concludes the proof of Assumption D.7.1.

Bibliography

- [1] ARM TrustZone. <http://www.arm.com/products/processors/technologies/trustzone.php>. URL <http://www.arm.com/products/processors/technologies/trustzone.php>. Accessed: 2017-04-08.
- [2] ARMv8 model. Available from: <http://www.cl.cam.ac.uk/~acjf3/l3/isa-models.tar.bz2>. Accessed: 2017-05-19.
- [3] seL4 Project. Available from: <http://sel4.systems/>. Accessed: 2017-04-21.
- [4] ARM security technology - building a secure system using trustzone technology. Technical documentation ARM PRD29-GENC-009492C. ARM Limited, 2009.
- [5] ARM Cortex-A15 MPCore processor - technical reference manual. Technical document ARM DDI 0438I. ARM Limited, 2011.
- [6] Onur Aciçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, pages 112–121. Springer-Verlag, 2006. ISBN 3-540-49496-0, 978-3-540-49496-6. URL http://dx.doi.org/10.1007/11935308_9.
- [7] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th Symposium on Principles of Programming Languages*, POPL '00, pages 40–53. ACM, 2000. ISBN 1-58113-125-9. URL <http://doi.acm.org/10.1145/325694.325702>.
- [8] Eyad Alkassar, Ernie Cohen, Mark A. Hillebrand, Mikhail Kovalev, and Wolfgang Paul. Verifying shadow page table algorithms. In Roderick Bloem and Natasha Sharygina, editors, *Formal Methods in Computer-Aided Design, 10th International Conference (FMCAD 2010)*, Lugano, Switzerland, October 2010. IEEE Computer Society.
- [9] Eyad Alkassar, Ernie Cohen, Mikhail Kovalev, and Wolfgang J. Paul. Verification of TLB virtualization implemented in C. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*,

- VSTTE'12, pages 209–224, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27704-7. URL http://dx.doi.org/10.1007/978-3-642-27705-4_17.
- [10] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load. *J. Autom. Reasoning*, 42(2-4):389–454, 2009. URL <http://dx.doi.org/10.1007/s10817-009-9123-z>.
 - [11] Eyad Alkassar, Mark A Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments*, pages 40–54. Springer, 2010.
 - [12] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, pages 163–184, 2016. URL http://dx.doi.org/10.1007/978-3-662-52993-5_9.
 - [13] Jim Alves-foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2:239–247, 2006.
 - [14] Jr. Ames, S. R., M. Gasser, and R. R. Schell. Security kernel design and implementation: An introduction. *Computer*, 16(7):14–22, July 1983. ISSN 0018-9162. URL <http://dx.doi.org/10.1109/MC.1983.1654439>.
 - [15] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.
 - [16] James P Anderson. Computer security technology planning study. volume 2. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.
 - [17] ARMv7-A architecture reference manual. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c>. Accessed: 2017-06-29.
 - [18] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the Conference on Computer and Communications Security*, CCS'14, pages 90–102. ACM, 2014.
 - [19] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin C Pierce, Randy Pollack,

- and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 165–178. ACM, 2014.
- [20] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer Dy, Javed Aslam, and David Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, July 2011. ISSN 0163-5980. URL <http://doi.acm.org/10.1145/2007183.2007189>.
 - [21] R. J. R. Back and J. von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In *Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, REX workshop, pages 42–66, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-52559-9. URL <http://dl.acm.org/citation.cfm?id=91930.91936>.
 - [22] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Åbo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A-1978-4.
 - [23] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998. ISBN 0387984178.
 - [24] Musard Balliu, Mads Dam, and Roberto Guanciale. Automating information flow analysis of low level code. In *Proceedings of the Conference on Computer and Communications Security*, CCS’14, pages 1080–1091. ACM, 2014. ISBN 978-1-4503-2957-6. URL <http://doi.acm.org/10.1145/2660267.2660322>.
 - [25] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI’11, pages 54–69, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18274-7. URL <http://dl.acm.org/citation.cfm?id=1946284.1946290>.
 - [26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
 - [27] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 1267–1279, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. URL <http://doi.acm.org/10.1145/2660267.2660283>.

- [28] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Jesús Mauricio Chimento, and Carlos Luna. Formally verified implementation of an idealized model of virtualization. In *Proceedings of the 19th International Conference on Types for Proofs and Programs*, TYPES'13, pages 45–63, 2014.
- [29] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Proc. FM'11*, volume 6664 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2011.
- [30] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proc. CSF'12*, pages 186–197, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4718-3. URL <http://dx.doi.org/10.1109/CSF.2012.17>.
- [31] Mick Bauer. Paranoid penguin: An introduction to Novell AppArmor. *Linux J.*, 2006(148):13–, August 2006. ISSN 1075-3583. URL <http://dl.acm.org/citation.cfm?id=1149826.1149839>.
- [32] C. Baumann, M. Näslund, C. Gehrmann, O. Schwarz, and H. Thorsen. A high assurance virtualization platform for armv8. In *2016 European Conference on Networks and Communications (EuCNC)*, pages 210–214, June 2016.
- [33] Christoph Baumann, Thorsten Bormer, Holger Blasum, and Sergey Tverdyshev. Proving memory separation in a microkernel by code level verification. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 25–32. IEEE, 2011.
- [34] Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk. Combining mechanized proofs and model-based testing in the formal analysis of a hypervisor. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 69–84, 2016. URL https://doi.org/10.1007/978-3-319-48989-6_5.
- [35] Muli Ben-yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, and Elsie Wahlig. Utilizing iommus for virtualization in linux and xen. In *In Proceedings of the Linux Symposium*, 2006.
- [36] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, ITCC'05, pages 586–591. IEEE Computer Society, 2005. ISBN 0-7695-2315-3. URL <http://dx.doi.org/10.1109/ITCC.2005.62>.

- [37] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, November 1989. ISSN 0098-5589. URL <http://dx.doi.org/10.1109/32.41331>.
- [38] William R. Bevier, William R. Bevier, and William R. Bevier. A verified operating system kernel, 1987.
- [39] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. URL <http://doi.acm.org/10.1145/1346281.1346286>.
- [40] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES’06, pages 201–215, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-46559-6, 978-3-540-46559-1. URL http://dx.doi.org/10.1007/11894063_16.
- [41] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540741127.
- [42] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017. URL <http://arxiv.org/abs/1702.07521>.
- [43] Sergey Bratus, Nihal D’Cunha, Evan Sparks, and Sean W. Smith. TOCTOU, traps, and trusted computing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Trust’08, pages 14–32. Springer-Verlag, 2008. ISBN 978-3-540-68978-2. URL http://dx.doi.org/10.1007/978-3-540-68979-9_2.
- [44] BillyBob Brumley. Cache storage attacks. In *Topics in Cryptology CT-RSA*, pages 22–34. 2015. ISBN 978-3-319-16714-5. URL http://dx.doi.org/10.1007/978-3-319-16715-2_2.
- [45] David Brumley. *Analysis and defense of vulnerabilities in binary code*. PhD thesis, Stanford University, 2008.
- [46] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM’03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=1251353.1251354>.

- [47] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *CAV*, pages 463–469, 2011.
- [48] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS ’01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society. URL <http://dl.acm.org/citation.cfm?id=874075.876409>.
- [49] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey DwoSkin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. URL <http://doi.acm.org/10.1145/1346281.1346284>.
- [50] Bill Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *In Proc. Winter USENIX Conference*, pages 163–174, 1992.
- [51] Hind Chfouka, Hamed Nemat, Roberto Guanciale, Mads Dam, and Patrik Ekdahl. Trustworthy prevention of code injection in linux on embedded devices. In *Proceedings of the 20th European Symposium on Research in Computer Security*, ESORICS’15, pages 90–107. Springer, 2015.
- [52] chroot. https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html. URL https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html. Accessed: 2017-04-12.
- [53] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. ISSN 0926-227X. URL <http://dl.acm.org/citation.cfm?id=1891823.1891830>.
- [54] D. L. Clutterbuck and B. A. Carre. The verification of low-level code. *Software Engineering Journal*, 3(3):97–111, May 1988. ISSN 0268-6961.
- [55] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on sel4. 2014.
- [56] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Proceedings of the Conference on Computer and Communications Security*, CCS’14, pages 570–581. ACM, 2014. ISBN 978-1-4503-2957-6. URL <http://doi.acm.org/10.1145/2660267.2660294>.
- [57] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying Concurrent C. In *Proc. TPHOLs’09*, pages 23–42, 2009.

- [58] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of multi core hypervisor verification. In *39th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'2013, pages 1–27. Springer, 2013.
- [59] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, pages 403–418, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14051-3, 978-3-642-14051-8. URL http://dx.doi.org/10.1007/978-3-642-14052-5_28.
- [60] Beagleboard. <http://beagleboard.org/>. URL <http://beagleboard.org/>. Accessed: 2017-04-08.
- [61] Coq. <https://coq.inria.fr/>. URL <https://coq.inria.fr/>. Accessed: 2017-04-08.
- [62] Cortex-A7 mpcore processors. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset.cortexa.cortexa7. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset.cortexa.cortexa7. Accessed: 2017-04-08.
- [63] Cortex-A8 processors. http://infocenter.arm.com/help/topic/com.arm.doc_subset.cortexa.a8. URL http://infocenter.arm.com/help/topic/com.arm.doc_subset.cortexa.a8. Accessed: 2017-04-08.
- [64] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
- [65] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for c and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 648–664, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. URL <http://doi.acm.org/10.1145/2908080.2908100>.
- [66] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy, Oakland*, volume 14, 2014.
- [67] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. URL <http://doi.acm.org/10.1145/2541940.2541986>.

- [68] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 83–100, Berkeley, CA, USA, 2009. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=1855768.1855774>.
- [69] John Criswell, Andrew Lenhardt, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. URL <http://doi.acm.org/10.1145/1294261.1294295>.
- [70] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
- [71] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 223–234. ACM, 2013.
- [72] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, pages 3–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2486-1. URL <http://doi.acm.org/10.1145/2517300.2517302>.
- [73] P Bovet Daniel and Cesati Marco. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.
- [74] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL <http://dl.acm.org/citation.cfm?id=1792734.1792766>.
- [75] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/360051.360056>.
- [76] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 431–446, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL <http://dl.acm.org/citation.cfm?id=2534766.2534804>.

- [77] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. *CoRR*, abs/1603.02187, 2016. URL <http://arxiv.org/abs/1603.02187>.
- [78] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the SMRAM: SMM reloaded. *CanSecWest*, 2009.
- [79] E. Allen Emerson. 25 years of model checking. chapter The Beginning of Model Checking: A Personal Perspective, pages 27–45. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69849-4. URL http://dx.doi.org/10.1007/978-3-540-69850-0_2.
- [80] Úlfar Erlingsson and Fred B Schneider. IRM enforcement of java stack inspection. Technical report, Ithaca, NY, USA, 2000.
- [81] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *IN PROCEEDINGS OF THE NATIONAL COMPUTER CONFERENCE*, pages 329–334. AFIPS Press, 1979.
- [82] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *proc. ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.
- [83] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, repairing, and verifying secvisor: A retrospective on the security of a hypervisor. *Tech. Rep. CMU-CyLab-08-008, Carnegie Mellon University*, 2008.
- [84] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.
- [85] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. URL <http://doi.acm.org/10.1145/945445.945464>.
- [86] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*, 2003. URL <http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf>.
- [87] GDB. <https://www.gnu.org/software/gdb/>. URL <https://www.gnu.org/software/gdb/>. Accessed: 2017-04-10.

- [88] Q. Ge, Y. Yarom, and G. Heiser. Do hardware cache flushing operations actually meet our expectations? *ArXiv e-prints*, December 2016.
- [89] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, -:1–27, dec 2016.
- [90] Michael Misiu Godfrey and Mohammad Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE T. Cloud Computing*, 2(4):395–408, 2014. URL <http://dx.doi.org/10.1109/TCC.2014.2358236>.
- [91] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*, pages 75–75, April 1984.
- [92] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
- [93] Hermann H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. 1947.
- [94] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09724-4. URL <http://dx.doi.org/10.1007/3-540-09724-4>.
- [95] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec’17, pages 2:1–2:6, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4935-2. URL <http://doi.acm.org/10.1145/3065913.3065915>.
- [96] David Greve, Matthew Wilding, and W Mark Vanfleet. A separation kernel formal security policy. In *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*. Citeseer, 2003.
- [97] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys’11, page 3. ACM, 2011.
- [98] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *ACM SIGPLAN Notices*, volume 50, pages 595–608. ACM, 2015.
- [99] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, pages 653–

- 669, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <http://dl.acm.org/citation.cfm?id=3026877.3026928>.
- [100] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 38–55, 2016. URL <http://dx.doi.org/10.1109/SP.2016.11>.
- [101] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *Journal of Computer Security*, 24(6):793–837, 2016. URL <http://dx.doi.org/10.3233/JCS-160558>.
- [102] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the Symposium on Security and Privacy, SP’11*, pages 490–505. IEEE Computer Society, 2011. ISBN 978-0-7695-4402-1. URL <http://dx.doi.org/10.1109/SP.2011.22>.
- [103] David S. Hardin. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1441915389, 9781441915382.
- [104] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel>.
- [105] Heartbleed website. <http://heartbleed.com/>. URL <http://heartbleed.com/>. Accessed: 2017-04-08.
- [106] Heartbleed wikipedia. <https://en.wikipedia.org/wiki/Heartbleed/>. URL <https://en.wikipedia.org/wiki/Heartbleed/>. Accessed: 2017-04-08.
- [107] 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*, pages 19–24. ACM, 2010.
- [108] Constance Heitmeyer, Myla Archer, Elizabeth Leonard, and John McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, January 2008. ISSN 0098-5589. URL <http://dx.doi.org/10.1109/TSE.2007.70772>.

- [109] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 346–355, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. URL <http://doi.acm.org/10.1145/1180405.1180448>.
- [110] Mark A Hillebrand, Thomas In der Rieden, and Wolfgang J Paul. Dealing with I/O devices in the context of pervasive system verification. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, ICCD'05, pages 309–316. IEEE, 2005.
- [111] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/363235.363259>.
- [112] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. URL <http://doi.acm.org/10.1145/2451116.2451146>.
- [113] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP Workshop on Programm Languages and Operating Systems*. Citeseer, 2005.
- [114] HOL4. <http://hol.sourceforge.net/>. URL <http://hol.sourceforge.net/>. Accessed: 2017-04-08.
- [115] Wei-Ming Hu. Reducing timing channels with fuzzy time. *J. Comput. Secur.*, 1(3-4):233–254, May 1992. ISSN 0926-227X. URL <http://dl.acm.org/citation.cfm?id=2699806.2699810>.
- [116] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.
- [117] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.
- [118] Mehmet Sinan Inci, Berk Gülmезoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm RSA key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015. URL <http://eprint.iacr.org/2015/898>.

- [119] Intel 64 and IA-32 Architectures Software Developer's Manual. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>. URL <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.
- [120] Intel. ISCA 2015 SGX tutorial. 2015. URL <https://software.intel.com/sites/default/files/332680-002.pdf>.
- [121] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.
- [122] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-vm attacks on xen and vmware. In *Proceedings of the 2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, BDCLOUD '14, pages 737–744, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6719-3. URL <http://dx.doi.org/10.1109/BDCLOUD.2014.102>.
- [123] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. URL <http://doi.acm.org/10.1145/1315245.1315262>.
- [124] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *ACSAC*, pages 119–126. IEEE Computer Society, 2002. ISBN 0-7695-1828-1. URL <http://dblp.uni-trier.de/db/conf/acsac/acsac2002.html#KargerS02>.
- [125] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *J. Comput. Secur.*, 8(2,3):141–158, August 2000. ISSN 0926-227X. URL <http://dl.acm.org/citation.cfm?id=1297828.1297833>.
- [126] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2Dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 957–972, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL <http://dl.acm.org/citation.cfm?id=2671225.2671286>.
- [127] Narges Khakpour, Oliver Schwarz, and Mads Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs*, pages 276–291. Springer, 2013.

- [128] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2Nd ACM Conference on Computer and Communications Security*, CCS '94, pages 18–29, New York, NY, USA, 1994. ACM. ISBN 0-89791-732-4. URL <http://doi.acm.org/10.1145/191177.191183>.
- [129] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=2362793.2362804>.
- [130] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. URL <http://dx.doi.org/10.1109/SP.2006.38>.
- [131] Gerwin Klein. From a verified kernel towards verified systems. In *proc. 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 21–33. Springer-Verlag, 2010.
- [132] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014. URL <http://doi.acm.org/10.1145/2560537>.
- [133] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dharmika Elkaduve, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220. ACM, 2009.
- [134] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113. Springer-Verlag, 1996. ISBN 3-540-61512-1. URL <http://dl.acm.org/citation.cfm?id=646761.706156>.
- [135] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 564–580, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0. URL http://dx.doi.org/10.1007/978-3-642-31424-7_40.
- [136] Eric Lacombe, Vincent Nicomette, and Yves Deswarthe. Enforcing kernel constraints by hardware-assisted virtualization. *Journal in Computer Virology*, 7

- (1):1–21, 2011. ISSN 1772-9890. URL <http://dx.doi.org/10.1007/s11416-009-0129-1>.
- [137] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974. ISSN 0163-5980. URL <http://doi.acm.org/10.1145/775265.775268>.
- [138] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. FM’09*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05088-6. URL http://dx.doi.org/10.1007/978-3-642-05089-3_51.
- [139] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL <http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf>.
- [140] H. Li, J. Zhu, T. Zhou, and Q. Wang. A new mechanism for preventing HVM-Aware malware. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 163–167, May 2011.
- [141] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and improving Linux kernel memory protection: a model checking approach. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 271–280. ACM, 2010.
- [142] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.
- [143] Lionel Litty, H Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.
- [144] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-10-3. URL <http://dl.acm.org/citation.cfm?id=647054.715771>.
- [145] Nancy Lynch and Frits Vaandrager. Forward and backward simulations i.: Untimed systems. *Inf. Comput.*, 121(2):214–233, September 1995. ISSN 0890-5401. URL <http://dx.doi.org/10.1006/inco.1995.1134>.
- [146] Michael McCoyd, Robert Bellarmine Krug, Deepak Goel, Mike Dahlin, and William Young. Building a hypervisor on a formally verifiable protection layer. In *Proceedings of the 2013 46th Hawaii International Conference on System Sciences*, HICSS ’13, pages 5069–5078, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4892-0. URL <http://dx.doi.org/10.1109/HICSS.2013.121>.

- [147] John McDermott, Bruce Montrose, Margery Li, James Kirby, and Myong Kang. Separation virtual machine monitors. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 419–428, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. URL <http://doi.acm.org/10.1145/2420950.2421011>.
- [148] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.
- [149] Marshall Kirk McKusick and George V Neville-Neil. *The design and implementation of the FreeBSD operating system*. Addison-Wesley Professional, 2004.
- [150] John McLean. Proving noninterference and functional correctness using traces. *J. Comput. Secur.*, 1(1):37–57, January 1992. ISSN 0926-227X. URL <http://dl.acm.org/citation.cfm?id=2699855.2699858>.
- [151] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=1268299.1268322>.
- [152] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP'12, pages 126–142, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-35307-9. URL http://dx.doi.org/10.1007/978-3-642-35308-6_12.
- [153] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, 2013.
- [154] Magnus O Myreen. Experience of tackling medium-sized case studies using decompilation into logic. *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications*, July 12-13 2014. URL http://cs.ru.nl/~freekver/acl2_14_slides/myreen_keynote.pdf.
- [155] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8, 2008. URL <http://dx.doi.org/10.1109/FMCAD.2008.ECP.24>.

- [156] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. URL <http://doi.acm.org/10.1145/503272.503286>.
- [157] Hamed Nemati, Roberto Guanciale, and Mads Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'15, pages 578–589. Springer, 2015. ISBN 978-3-662-46078-8. URL http://dx.doi.org/10.1007/978-3-662-46078-8_48.
- [158] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, SAC'06, pages 147–162. Springer-Verlag, 2007. ISBN 978-3-540-74461-0. URL <http://dl.acm.org/citation.cfm?id=1756516.1756531>.
- [159] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.
- [160] Peter Norvig. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. 2006. URL <https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>. Accessed: 2017-04-08.
- [161] OpenSSL. <https://www.openssl.org>. URL <https://www.openssl.org>.
- [162] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and counter-measures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-31033-9, 978-3-540-31033-4. URL http://dx.doi.org/10.1007/11605805_1.
- [163] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.
- [164] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In *Proc. SEFM'12*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer Berlin Heidelberg, 2012.
- [165] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. URL <http://dx.doi.org/10.1109/SP.2008.24>.

- [166] Colin Percival. Cache missing for fun and profit. *BSDCan*, 2005.
- [167] Danny Philippe-Jankovic and Tanveer Zia. Breaking vm isolation: An in-depth look into the cross flush reload cache timing attack. *International Journal of Computer Science and Network Security*, 17(2):181–193, 2017. ISSN 1738-7906. Imported on 12 Apr 2017 - DigiTool details were: Journal title (773t) = International Journal of Computer Science and Network Security. ISSNs: 1738-7906;;
- [168] Gordon Plotkin, Colin Stirling, and Mads Tofte, editors. *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5.
- [169] Power ISA version 2.07. https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf. URL https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf. Accessed: 2017-04-08.
- [170] Reduce resolution of performance.now to prevent timing attacks. <https://bugs.chromium.org/p/chromium/issues/detail?id=506723/>, 2015. URL <https://bugs.chromium.org/p/chromium/issues/detail?id=506723/>.
- [171] Ning Qu, XG Gou, and Xu Cheng. Using uncacheable memory to improve unity linux performance. In *Proceedings of the 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 27–32, 2005.
- [172] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the Workshop on Cloud Computing Security*, CCSW ’09, pages 77–84. ACM, 2009. ISBN 978-1-60558-784-4. URL <http://doi.acm.org/10.1145/1655008.1655019>.
- [173] Alastair Reid. Trustworthy specifications of arm® v8-a and v8-m system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD ’16, pages 161–168, Austin, TX, 2016. FMCAD Inc. ISBN 978-0-9835678-6-8. URL <http://dl.acm.org/citation.cfm?id=3077629.3077658>.
- [174] RaymondJ. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010. ISBN 978-1-4419-1538-2. URL http://dx.doi.org/10.1007/978-1-4419-1539-9_10.
- [175] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the*

- 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. URL http://dx.doi.org/10.1007/978-3-540-87403-4_1.
- [176] Lawrence Robinson, Karl N Levitt, Peter G Neumann, and Ashok R Saxena. A formal methodology for the design of operating system software. *Current Trends in Programming Methodology*, 1:61–110, 1977.
 - [177] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1st edition, 2008. ISBN 9780521103503.
 - [178] Mendel Rosenblum. Vmwares virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.
 - [179] Raspberry Pi 2 Model B. <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>. URL <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>. Accessed: 2017-04-08.
 - [180] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM. ISBN 0-89791-062-1. URL <http://doi.acm.org/10.1145/800216.806586>.
 - [181] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992. URL <http://www.csl.sri.com/papers/csl-92-2/>.
 - [182] Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, page 54, 2010.
 - [183] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Doorn, John Linwood, and Griffin Stefan Berger. sHype: Secure hypervisor approach to trusted virtualized systems. In *IBM Research Report RC23511*, 2005.
 - [184] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8. URL <http://dl.acm.org/citation.cfm?id=647348.724331>.
 - [185] Edward J. Schwartz. *Abstraction Recovery for Scalable Static Binary Analysis*. PhD thesis, Carnegie Mellon University, 2014.
 - [186] Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with DMA. In *Hardware and Software: Verification and Testing*, pages 236–251. Springer, 2014.

- [187] Oliver Schwarz and Mads Dam. Automatic derivation of platform non-interference properties. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, pages 27–44, 2016. URL https://doi.org/10.1007/978-3-319-41591-8_3.
- [188] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium*, pages 1–12, 2010.
- [189] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP ’07*, pages 335–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. URL <http://doi.acm.org/10.1145/1294261.1294294>.
- [190] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13*, pages 471–482, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. URL <http://doi.acm.org/10.1145/2491956.2462183>.
- [191] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS ’07*, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. URL <http://doi.acm.org/10.1145/1315245.1315313>.
- [192] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS ’09*, pages 477–487, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. URL <http://doi.acm.org/10.1145/1653662.1653720>.
- [193] Di Shen. Exploiting Trustzone on Android. *Black Hat US*, 2015. URL <https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf/>.
- [194] Bruno R Silva, Diego Aranha, and Fernando MQ Pereira. Uma técnica de análise estática para detecc. In *In Brazilian Symposium on Information and Computational Systems Security*, pages 16–29, Florianopolis, SC, BR, December 2015.
- [195] Scott M. Silver. Implementation and analysis of software based fault isolation. Technical Report PCS-TR96-287, Dartmouth College, 1996.

- [196] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89861-0. URL http://dx.doi.org/10.1007/978-3-540-89862-7_1.
- [197] Evan R Sparks and Evan R Sparks. A security assessment of trusted platform modules computer science technical report tr2007-597. *Dept. Comput. Sci., Dartmouth College, Hanover, NH, USA, Tech. Rep.*, TR2007-597, 2007.
- [198] Deian Stefan, Pablo Buiras, EdwardZ. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS'13, pages 718–735. Springer, 2013. ISBN 978-3-642-40202-9.
- [199] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222. ACM, 2010. ISBN 978-1-60558-577-2. URL <http://doi.acm.org/10.1145/1755913.1755935>.
- [200] Disakil malware. <https://www.symantec.com/connect/blogs/destructive-disakil-malware-linked-ukraine-power-outages-also-used-against-media-organizations>. Accessed: 2017-04-08.
- [201] One New Zero-Day Discovered on Average Every Week in 2015. <https://www.symantec.com/security-center/threat-report>. URL <https://www.symantec.com/security-center/threat-report>.
- [202] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 352–367, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28584-9, 978-3-540-28584-7. URL http://dx.doi.org/10.1007/11547662_24.
- [203] Hendrik Tews, Marcus Völp, and Tjark Weber. Formal memory models for the verification of low-level operating-system code. *J. Autom. Reasoning*, 42(2-4):189–227, 2009. URL <http://dx.doi.org/10.1007/s10817-009-9122-0>.
- [204] Hendrik Tews, Tjark Weber, and Marcus Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. *Electron. Notes Theor. Comput. Sci.*, 217:79–96, July 2008. ISSN 1571-0661. URL <http://dx.doi.org/10.1016/j.entcs.2008.06.043>.

- [205] The BeagleBoard.org Foundation. The BeagleBoard-xM, 2010. URL <http://beagleboard.org/Products/BeagleBoard-xM>. <https://www.wolfssl.com/wolfSSL/Home.html>.
- [206] Julien Tinnes. Linux null pointer dereference due to incorrect proto-ops initializations (cve-2009-2692), 2009. URL <http://blog.cr0.org/2009/08/linux-null-pointer-dereference-due-to.html>.
- [207] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. URL <http://doi.acm.org/10.1145/2000064.2000087>.
- [208] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71, January 2010. ISSN 0933-2790. URL <http://dx.doi.org/10.1007/s00145-009-9049-y>.
- [209] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, CHES'03, LNCS, pages 62–76. Springer, 2003.
- [210] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1179-3. URL <http://doi.acm.org/10.1145/2103799.2103813>.
- [211] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. URL <http://dx.doi.org/10.1109/SP.2013.36>.
- [212] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 41–46, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1004-8. URL <http://doi.acm.org/10.1145/2046660.2046671>.
- [213] Verisoft project. <http://www.verisoft.de>. URL <http://www.verisoft.de>.
- [214] Verisoft-XT project. <http://www.verisoftxt.de/>. URL <http://www.verisoftxt.de/>.

- [215] Bárbara Vieira. *Formal verification of cryptographic software implementations*. PhD thesis, Universidade do Minho, Portugal, 2012. URL <https://repositorium.sdum.uminho.pt/bitstream/1822/20770/1/B%C3%A1rbara%20Isabel%20de%20Sousa%20Vieira.pdf>.
- [216] David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *Computer Security - ESORICS 2004, 9th European Symposium on Research Computer Security, Sophia Antipolis, France, September 13-15, 2004, Proceedings*, pages 225–243, 2004. URL https://doi.org/10.1007/978-3-540-30108-0_14.
- [217] David A. Wagner. Janus: An approach for confinement of untrusted applications. Technical report, Berkeley, CA, USA, 1999.
- [218] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP ’93, pages 203–216, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8. URL <http://doi.acm.org/10.1145/168619.168635>.
- [219] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix Security Kernel. *Commun. ACM*, 23(2):118–131, February 1980. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/358818.358825>.
- [220] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 260–275, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. URL <http://doi.acm.org/10.1145/2517349.2522728>.
- [221] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA ’07, pages 494–505, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. URL <http://doi.acm.org/10.1145/1250662.1250723>.
- [222] Zhi Wang and Xuxian Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. URL <http://dx.doi.org/10.1109/SP.2010.30>.
- [223] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First USENIX Workshop on Offensive*

- Technologies*, WOOT'07, pages 2:1–2:8. USENIX Association, 2007. URL <http://dl.acm.org/citation.cfm?id=1323276.1323278>.
- [224] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. Proceedings of the 16th international conference on financial cryptography and data security. FC'2012, pages 314–328. Springer, 2012. ISBN 978-3-642-32946-3. URL http://dx.doi.org/10.1007/978-3-642-32946-3_23.
- [225] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer US, 2010. ISBN 978-1-4419-1538-2. URL http://dx.doi.org/10.1007/978-1-4419-1539-9_6.
- [226] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via intel CPU cache poisoning. *Invisible Things Lab*, 2009.
- [227] wolfSSL: Embedded SSL Library. <https://www.wolfssl.com/wolfSSL/Home.html>. URL <https://www.wolfssl.com/wolfSSL/Home.html>. Accessed: 2017-04-08.
- [228] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-00-5. URL <http://dl.acm.org/citation.cfm?id=647253.720287>.
- [229] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 414–425, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. URL <http://doi.acm.org/10.1145/2810103.2813637>.
- [230] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL <http://dl.acm.org/citation.cfm?id=2671225.2671271>.
- [231] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110. ACM, 2012. ISBN 978-1-4503-1205-9. URL <http://doi.acm.org/10.1145/2254064.2254078>.

- [232] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vmm side channels and their use to extract private keys. In *Proceedings of the Conference on Computer and Communications Security*, CCS '12, pages 305–316. ACM, 2012. ISBN 978-1-4503-1651-4. URL <http://doi.acm.org/10.1145/2382196.2382230>.
- [233] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: Fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 289–298. IEEE, 2011.
- [234] Xin-Jie Zhao and Tao Wang. Improved cache trace attack on AES and CLE-FIA by considering cache miss and S-box misalignment. *IACR Cryptology ePrint Archive*, 2010:56, 2010.
- [235] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. High-assurance separation kernels: A survey on formal methods. *CoRR*, abs/1701.01535, 2017. URL <http://arxiv.org/abs/1701.01535>.
- [236] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 871–882, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. URL <http://doi.acm.org/10.1145/2976749.2978324>.