KTH Computer Science
and Communication

No Hypervisor Is an Island: System-wide Isolation Guarantees for Low Level Code

OLIVER SCHWARZ

Doctoral Thesis
Stockholm, Sweden 2016

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

TRITA-CSC-A 2016:22
ISSN 1653-5723
ISRN-KTH/CSC/A--16/22-SE
ISBN 978-91-7729-104-6

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläggges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi måndagen den 10 oktober 2016 klockan 14.00 i F3, Kungl Tekniska högskolan, Lindstedtsvägen 26, Stockholm.

SICS Swedish ICT
SICS Dissertation Series 75
ISSN 1101-1335

Cover picture: The photograph that serves as background of the cover picture is courtesy of Lukáš Poláček.

Tryck: Universitetsservice US AB

Abstract

The times when malware was mostly written by curious teenagers are long gone. Nowadays, threats come from criminals, competitors, and government agencies. Some of them are very skilled and very targeted in their attacks. At the same time, our devices – for instance mobile phones and TVs – have become more complex, connected, and open for the execution of third-party software. Operating systems should separate untrusted software from confidential data and critical services. But their vulnerabilities often allow malware to break the separation and isolation they are designed to provide. To strengthen protection of select assets, security research has started to create complementary machinery such as security hypervisors and separation kernels, whose sole task is separation and isolation. The reduced size of these solutions allows for thorough inspection, both manual and automated. In some cases, formal methods are applied to create mathematical proofs on the security of these systems.

The actual isolation solutions themselves are carefully analyzed and included software is often even verified on binary level. The role of other software and hardware for the overall system security has received less attention so far. The subject of this thesis is to shed light on these aspects, mainly on (i) unprivileged third-party code and its ability to influence security, (ii) peripheral devices with direct access to memory, and (iii) boot code and how we can selectively enable and disable isolation services without compromising security.

The six papers included in this thesis are both design and verification oriented, however, with an emphasis on the analysis of instruction set architectures. With the help of a theorem prover, we implemented various types of machinery for the automated information flow analysis of several processor architectures. We used these tools to make explicit which registers arbitrary and unprivileged software on ARM or MIPS platforms can access. The analysis is guaranteed to be both sound and accurate. To the best of our knowledge, we were the first to publish an automated analysis and verification of information flow properties for commodity instruction set architectures.

Sammanfattning

Förr skrevs skadlig mjukvara mest av nyfikna tonåringar. Idag är våra datorer under ständig hot från statliga organisationer, kriminella grupper, och kanske till och med våra affärskonkurrenter. Vissa besitter stor kompetens och kan utföra fokuserade attacker. Samtidigt har tekniken runtomkring oss (såsom mobiltelefoner och tv-apparater) blivit mer komplex, uppkopplad och öppen för att exekvera mjukvara från tredje part.

Operativsystem borde egentligen isolera känslig data och kritiska tjänster från mjukvara som inte är trovärdig. Men deras sårbarheter gör det oftast möjligt för skadlig mjukvara att ta sig förbi operativsystemens säkerhetsmekanismer. Detta har lett till utveckling av kompletterande verktyg vars enda funktion är att förbättra isolering av utvalda känsliga resurser. Speciella virtualiseringsmjukvaror och separationskärnor är exempel på sådana verktyg. Eftersom sådana lösningar kan utvecklas med relativt liten källkod, är det möjligt att analysera dem noggrant, både manuellt och automatiskt. I några fall används formella metoder för att generera matematiska bevis på att systemet är säkert.

Själva isoleringsmjukvaran är oftast utförligt verifierad, ibland till och med på assemblernivå. Dock så har andra komponenters påverkan på systemets säkerhet hittills fått mindre uppmärksamhet, både när det gäller hårdvara och annan mjukvara. Den här avhandlingen försöker belysa dessa aspekter, huvudsakligen (i) oprivilegierad kod från tredje part och hur den kan påverka säkerheten, (ii) periferienheter med direkt tillgång till minnet och (iii) startkoden, samt hur man kan aktivera och deaktivera isolationstjänster på ett säkert sätt utan att starta om systemet.

Avhandlingen är baserad på sex tidigare publikationer som handlar om både design- och verifikationsaspekter, men mest om säkerhetsanalys av instruktionsuppsättningar. Baserat på en teorembevisare har vi utvecklat olika verktyg för den automatiska informationsflödesanalysen av processorer. Vi har använt dessa verktyg för att tydliggöra vilka register oprivilegierad mjukvara har tillgång till på ARM- och MIPS-maskiner. Denna analys är garanterad att vara både korrekt och precis. Så vitt vi vet är vi de första som har publicerat en lösning för automatisk analys och bevis av informationsflödesegenskaper i standardinstruktionsuppsättningar.

Acknowledgements

No person is an island. Especially no PhD student. In my case, they even gave me two lands that took care of me: the SICS-land and the KTH-land. Both came with one advisor each and – maybe without being aware of it – the two complemented each other pretty well.

My industrial advisor Christian Gehrmann understood to accompany my development with a supervision ever-aligned with the needs of my respective state. In the beginning he provided much initial guiding, followed by an increasing amount of trust, freedom, and responsibility – always on the right level, always *lagom mycket* to empower me. It is not granted for industrial PhD students to find that degree of freedom and time for research and studies that I received from Christian. He always cared about my progress and expressed his confidence in me. Thank you very much, Christian!

My gratitude also goes to my academic advisor Mads Dam. He always found time and patience to discuss technical details or writing matters. And this is still an understatement. As for technical details, it is more precise to say that Mads taught me to never stop discussing them until I really understand their gist. And as for writing, he went through uncountable iterations of proofreading with me. As much as I was striving towards his level of writing, I did not even come close and he still remains a role model in that respect. I also want to thank Mads for the freedom he granted me in my research questions and his confidence in the paths I have chosen towards their solutions.

Also, I want to thank Mads for creating a truly inspirational research environment. The PROSPER group at KTH was always a place to discuss, receive help, and get inspired. Working together with that group really was a pleasure. I want to thank their members, both my co-authors and the rest, namely Mads, Narges Khakpour, Hamed Nemati, Christoph Baumann, Andreas Lindner, Dilian Gurov, Musard Balliu, Andreas Lundblad, and Roberto Guanciale. Roberto, I am sorry that I do not know how to put this in a less stereotypical way, but you are a true role model, both as a scientist and a person.

Thanks go also to my co-authors and project partners at SICS, namely Christian, Viktor Do, Arash Vahidi, Heradon Douglas, and Jonas Haglund. Especially in the early days, you were my tour guides through the – back then for me inscrutable – jungle of low-level hardware and system software matters. Whenever I had a hardware or implementation question, you were the right address to look for the answer. I also want to thank the other members of the SEC lab, Rolf Blom, Rikard Höglund, Marco Tiloca, Nicolae Paladi, Antonis Michalas, Mudassar Aslam, Ludwig Seitz, our master students, and all our lab members in Lund, that I hopefully will get to know better in the future. Nicolae, thank you for proofreading so many of my papers – even though I use parentheses more often than your beloved dashes. Thank you also for all the interesting scientific discussions in your office. The rubber duck, the whiteboard pictures, and that strange little spinning tops created a pleasant and stimulating atmosphere for brainstorming. Marco, thank you for sharing your insights about the world, both in scientific and other respects. Mudassar, the same holds for you. It was great to share the office with you, share our views on being PhD students, and share our thoughts concerning the big questions of life.

Back in KTH-land, they also put me into an office, specifically, into the best office of the entire KTH. Thank you Benjamin Greschbach, Emma Enström, and Guillermo Rodríguez Cano for having kept it such a welcoming place that always encouraged people to drop by. Another place at the TCS-department that I never will forget is its kitchen. It was the home for countless interesting lunch discussions. Many people deserve thanks for this, but in particular Gunnar Kreitz, Lukáš Poláček, and Lukáš' mother. There are many more nice people at TCS. I like to thank them all for a nice working environment. I know these acknowledgements will be the most read part of the thesis and I know you are waiting for your names to be listed here. But I have been around for quite many years, so many people came and went, just too many to list all of you. But be sure that you are in my memories, nonetheless. The same goes for all the nice people at SICS. I cannot list you all, but I really enjoyed your company and help. Thanks to the old NETS lab, to the IT support, the receptionists, the administration, the Swedish teachers, the badminton partners, and all others. The heterogeneity of the people at SICS, of gender, age, profession, and origin creates the feeling of a big family that I really enjoyed.

Special thanks go to Mudassar, Benjamin, Nicolae, and Hamed. They have been there in the hard times that such an enterprise sometimes has. Thanks for comfort and encouragement and just being there as friends.

Thanks to all other people that I met on the way, all project partners, all interesting people I got to know at conferences and summer schools, and all anonymous reviewers that helped to improve my papers.

Thank you, Gerwin Klein, Mads, Karl Meinke, Nicolae, Christoph, Deborah Fauser, Rikard, Arash, and Jonas, for reviewing and commenting this thesis, be it a few lines or many many pages. I really appreciate your help.

I also want to thank all residents of Villa Leipzig, especially Barbro and Robert, but also the other students and further tenants, for providing me not only with accommodation, but also with a true home during my time as PhD student.

Many thanks go to my friends in Sweden, Germany, and all over the world. You have grounded me throughout these years. Thank you, Deborah for your patience and support. You allowed me to recharge on long working days. Vielen lieben Dank an meine Familie, insbesondere meine Eltern! Hättet ihr damals nicht die Besonnenheit gehabt, die offizielle Bildungsempfehlung zu ignorieren, wäre diese Doktorarbeit vermutlich nie zustande gekommen. Ihr fahrt durch halb Europa, um mir zu helfen, und seid auch sonst stets für mich da. Vielen Dank!

Contents

Part I

Thesis

Acronyms

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

ARM ARM ARM Architecture Reference Manual. 32

BAP the Binary Analysis Platform. 17

BIOS basic input/output system. 15

CC Common Criteria. 28

DMA direct memory access. 14

DMAC direct memory access controller. 39

EAL evaluation assurance level. 28

GPU graphics processing unit. 29

HDD hard disk drive. 15

IOMMU input/output memory management unit. 7, 15

IPC inter-process communication. 26

ISA instruction set architecture. 13

MMU memory management unit. 10

MPU memory protection unit. 10

NIC network interface controller. 15

OHCI Open Host Controller Interface. 15

OS operating system. 9

REE rich execution environment. 10

RISC reduced instruction set computing. 29

SGX Software Guard Extensions. 11

SML Standard ML. 18

SMM System Management Mode. 15

SMMU system memory management unit. 15

SMT satisfiability modulo theories. 17

SoC System-on-a-Chip. 29

TCB trusted computing base. 5

TEE trusted execution environment. 10

TLB translation lookaside buffer. 30

TLS topl-level specification. 27

TPM Trusted Platform Module. 10

Chapter 1

Introduction

Encryption works. Properly implemented strong crypto systems are one of the few things that you can rely on. **Unfortunately, endpoint security is so terrifically weak that NSA can frequently find ways around it.**

Edward Snowden

In the 1990s I possessed an electronic organizer for teenagers. One of its features was the ability to send messages to other such organizers via infra-red transmission. According to the manufacturer, messages would be secret and only visible to me and the person I had a crush on. Back then I was not interested in security yet. Besides, none of the persons I had a crush on was an owner of such an organizer. Twenty years later I wonder how the manufacturer actually implemented its security promise. In good faith in the manufacturer, I assume they used the strongest encryption available. But as pointed out by Snowden in the quote above or by Kocher in [107], cryptography is less of a concern than the security of the computing devices on which this cryptography is processed. My electronic organizer at least ran some user interface, the encryption service, and the infra-red network driver, as shown in Figure 1.1. One might think that the encryption service is the only process relevant for security. However, the user interface had access to my love letter in clear text and possibly could have circumvented the encryption by sending the message to the infra-red module directly. Also, the network driver – maybe third party code – might have contained bugs or backdoors that somehow would read the love letter from the memory of the other modules. In the end, it is possible that confidentiality depended on all three processes. In technical terms, the *trusted computing base (TCB)* of my organizer might have included all of them.

If we are more optimistic, the electronic organizer might have contained some isolation mechanism (see Figure 1.2), that made sure that the modules were sepa-

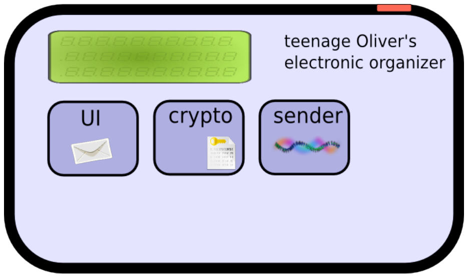

Figure 1.1: The electronic organizer with some of the software modules executing alongside each other.

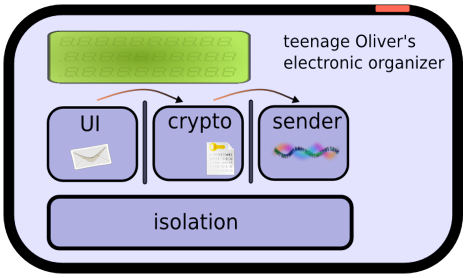

Figure 1.2: The electronic organizer with an isolation mechanism, separating the modules.

rated from each other and could not access each other's memory, except for some dedicated and controlled inter-process communication. With working isolation and the right policies in place, the TCB would then only have contained the encryption service and the isolation mechanism itself. Even if both user interface and network module were corrupted, my messages would always be encrypted before leaving the organizer.

For teenagers nowadays, life is even harder than it was for me back then. Leaving aside the many possible non-technical reasons, this is because their electronic organizers – commonly called smartphones – do not only run code from the manufacturer. Smartphones are connected to the Internet and allow the execution of third party programs. It is no longer only erroneous code that can threaten security, entire programs might be developed with malicious intents. This applies not solely to smartphones, but to many other connected devices, as well. Isolation is needed to protect assets from such malware. If attacks still succeed, isolation can help to limit the damage. And as we have seen above, isolation allows to reduce the TCB.

But in turn, a reduced TCB makes it also easier to increase confidence in the isolation. The smaller the isolation mechanism in my electronic organizer, the more feasible it was for the manufacturer to inspect the mechanism and clear it from bugs. To develop that line of thought a little further, it is worthwhile noticing that isolation has been a key feature of most computing systems for a long time. In fact, one of the tasks of an operating system is to allow the parallel or interleaved execution of several processes and to provide isolation by making sure that the processes do not interfere with each other outside regulated inter-process communication. However, many operating systems are vulnerable to attacks that threaten this isolation. A recent example that was covered broadly in media comprises three iOS vulnerabilities that together allow an attacker to silently install sophisticated surveillance software on the victim's phone when the victim clicks on a specific link [28]. Many more vulnerabilities – also for other systems – were reported in the past.

For performance and other reasons the design of most commodity operating systems is monolithic, leading to a large TCB. Assuming a constant error density, a larger code base leads to a higher number of errors. Furthermore, it increases the costs for inspection. In contrast, the analysis of isolation mechanisms with smaller and less complex implementations is easier. This is true for both manual inspection and computer-aided formal verification, that is, the generation of mathematical proofs on the security of a system.

Often such analysis focuses only on the isolating software. However, isolation depends on hardware, as well. Some of a system's hardware was built to enable isolation, such as memory protection units or virtualization extensions. Other hardware can threaten isolation, such as peripherals with direct memory access. It is important to consider these influences when designing and verifying isolation solutions, even more since the role of hardware for isolation is increasing. Similarly, it is crucial to keep in mind that the software responsible for isolation is not the only software on the system. Other software such as arbitrary user programs might compromise isolation if the underlying hardware platform is misconfigured or misdesigned. Furthermore, there might be software executing before the isolation software, potentially preventing isolation from ever being established.

The purpose of this thesis is to improve assurance in respect to such aspects – system aspects which constitute the environment that isolation software executes in. We are particularly interested in the context of virtualization as enabler for isolation, but many results are applicable to other system software, as well. While the thesis cannot address all open challenges, it attempts to contribute towards the ideal of pervasive platform security for general purpose systems. To that end, the following challenges are addressed:

- How can we analyze instruction set architectures to learn about possible information flows that can occur during unprivileged execution? Verifying isolation properties of the processor is more than just obtaining assurance of its security. It is also about learning what system components need to be banked, cleared, or restored on context switches. And finally, such analysis yields information on how to configure the system, such that user processes are sufficiently restricted.

- How can we maintain isolation when peripherals execute in parallel with the CPU and have direct access to the memory? A complete exclusion of such peripherals is not a practical option in most cases. Virtualization software needs to provide its guests with access to those devices, however, in a controlled manner. If no dedicated hardware support (such as an IOMMU, see Section 2.1.3) is available, the virtualization software needs to function as a proxy, check policies, and possibly multiplex accesses to peripherals. System models need to be adopted to enable reasoning on the partitioning implemented by the composed computing system.

- How do we guarantee the launch of uncompromised isolation software in previously unprotected software stacks without completely rebooting the system?

The thesis is comprised of both publications that focus on formal verification and publications that focus on the design of platform security solutions. While the work presented in this thesis was performed in projects with emphasis on embedded systems, many results are general and can be applied to strengthen the security of other platforms.

Thesis Outline Following this introduction, a background section establishes the main concepts within platform security and formal verification that are relevant for this thesis. Furthermore, the main challenges and related work are discussed. In Chapter 3 the contributions of the thesis are first described on a high level and then on a per-paper basis, together with declarations of the individual contributions of the author. For completeness, the author's publications outside this thesis are listed at the end of the chapter. Chapter 4 concludes the first part of the thesis with a summary and a discussion of possible future work. Thereafter, six of the author's paper are included. Some of them contain minor extensions to the original publications. Otherwise the papers are included as originally published, but adopted to the style format of this thesis. A common reference list for all included papers and introductory part of the thesis can be found at the end.

Chapter 2

Background

2.1 Platform Security

IT security can be divided into a number of subfields, including for instance cryptography or network security. This thesis focuses on the subfield of platform security. The goal of platform security is to secure computing platforms, ranging from small sensor nodes over smartphones and personal computers to industrial servers. The main ambition is to protect the integrity and confidentiality of software, credentials, and other data, while they reside on the computing platform. Threats to those security properties can be of both physical and logical nature. While physical attacks on the platforms have high impact potential, they are less likely in most scenarios. Instead, this thesis discusses protection from threats that allow malicious software (called *malware*) to infiltrate the victim's system via storage media or remotely via a network.

Two concepts are vital for protection from malware: *isolation* between potentially malicious software and assets to protect, and the establishment of *trust* in integrity and authenticity. Processes should not be allowed to influence other processes or access their data, except for explicit communication. Besides protecting processes from each other, the isolation enabler also needs to protect itself from undesired modification. Once a software stack with such isolation properties is developed, the second challenge is to provide the users or third parties with assurance that the system they interact with actually is the system they assume, that is, that they trust. This trust establishment is sometimes also called *attestation*. *Remote attestation* is the attestation of a system's state to a remote party, for instance over the network.

Isolation can be achieved in many ways. Software aiming to isolate processes (e.g., an operating system (OS) isolating applications) could potentially interpret the processes line by line and dynamically check each instruction against policies before executing it. Alternatively, it could statically check that code follows the desired policy before handing over execution to the corresponding processes. However, while the first option is prohibitive in terms of processing time, the second is hard to achieve, due to obstacles such as self-modifying code. In practice, system software therefore relies on hardware support to establish isolation. Such support allows direct execution of untrusted code on the CPU while still maintaining isolation. Most prominently, memory protection units (MPUs) and memory management units (MMUs) allow privileged code to set selected regions of the main memory inaccessible throughout the execution of a process. Memory protection policies can restrict readability, writability, executability, or combinations thereof. Processors usually support the execution on different privilege levels to make sure that applications cannot change memory protection or other privileged settings, while operating system kernels can. Transitions between privilege levels (or *operation modes*) follow well-defined schemes, that – for instance – enforce the execution of specific code (called *handlers*) on entrance into a privileged mode. The hardware can be configured to invoke the kernel on certain events (*exceptions*) such as interrupts from a peripheral or when unprivileged software attempts to access restricted memory.

2.1.1 Trusted Execution Environments

Due to the size and complexity of operating systems, OS developers often fail in keeping code free from vulnerabilities that threaten the isolation of user processes. Therefore, several initiatives attempt to enhance platforms with additional isolation that is stronger than the one provided by most operating systems. This isolation can then be used to protect select code and data, even if the OS is compromised. Environments providing such integrity- and confidentiality-protected execution are often referred to as *trusted execution environments (TEEs).* [1] They operate alongside the *rich execution environment (REE)* – including the commodity OS and its applications – but subject to some (hardware) separation [121] and usually offering services to the REE [136].

Some TEEs are provided by the processor designers, such as by ARM's TrustZone extension [11]. TrustZone's TEE is called *Secure World* and manages its own virtual MMU and own vector table [59]. An additional address bit allows to assign both memory regions and peripherals exclusively to the Secure World. However, the Secure World always has access to the REE (or *Normal World*).

Trusted computing [174] is another enabler for TEEs. In trusted computing, software that is loading new software uses special CPU instructions to calculate integrity measurements of the loaded binaries. Those measurements are computed with the help of hash functions and reported to a special cryptographic hardware component – the *Trusted Platform Module (TPM)* – before execution is handed over to the loaded software. Measurements can be computed incrementally while

[1] Note that the term TEE is overloaded in the security community. The definitions which we follow in this thesis only require isolated processing and memory. However, there are other definitions of TEEs that have stricter requirements and include, for example, boot integrity, secure storage, and device identification/authentication [63].

building up the software stack or only for a single binary block to load, sometimes called *late launch* [100, 82, 1, 116]. In the incremental case, the reported measurements help to attest the integrity of the entire software stack as long as all reporting modules are trusted. In late launch, hardware guarantees that the TPM-reported binary actually has received control at some point. Note that this attestation of integrity is not the same as integrity enforcement – it is still possible that manipulated binaries are loaded and executed. However, in that case attestation will fail. Since it is possible to make access to credentials subject to attested integrity, attacks on the software's integrity would be rendered useless in some scenarios. In practice, credentials are either provided from a remote machine after integrity has been attested or – with *sealed storage* – revealed by special hardware.

When those methods are used correctly, they can guarantee the integrity of launched software and indirectly the integrity and confidentiality of data. However, by itself this guarantee only holds as long as the launched software is the only one executing on the machine. When handing over control or when executing on multicore systems the software needs to take additional measures to maintain isolation, for instance through memory protection or clearing state on context switches.

To minimize such overhead - and in consequence the code to rely on - Intel introduced its *Software Guard Extensions (SGX)* [8, 119, 94, 118, 183]. TEEs based on SGX are called *enclaves* and it is the processor that ensures integrity and confidentiality once code and data are loaded into such an enclave. Throughout the execution of an enclave, software executing alongside cannot interfere with or learn about the enclave's execution, apart from explicit communication and side channels such as page faults (cf. [184]). Since enclave code is not protected before the enclave is loaded or after it is destroyed, remote attestation and sealed storage are employed to maintain protection of integrity and confidentiality at those times. For low-end embedded systems such as sensor nodes, Sancus [131] provides similar protection to that of SGX.

To fully understand the benefit of TEEs, the concept of a *trusted computing base (TCB)* is essential. The TCB of a system is "a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security" [109, page 270]. While other definitions (e.g., [110, page 112, glossary]) understand the TCB as a set of dedicated protection mechanisms, in this thesis TCB is to be understood as the totality of *any* software or hardware that *actually* has the capability and privileges (but not necessarily the intended functionality) to compromise security. The user thus needs to trust the TCB. Trust in that sense does not distinguish between malicious intentions and exploitable errors. For the user of a platform it does not really matter if the platform's operating system was written to leak/manipulate the user's data or if the operating system is "only" vulnerable enough to allow an attacker in the form of a malicious application or remote party to cause damage. Both cases are clearly undesirable. Since complex (sub-)systems are hard to secure, one of the main paradigms of security research has therefore become to minimize this trusted computing base. And that is in the end what TEEs are developed for. It is not the case that operating systems *in principle* could not isolate critical from untrusted applications. But isolation is not their sole task and – especially monolithic – operating systems often have a larger TCB than TEEs. This is clearly a disadvantage when it comes to exhaustive examination.

We next turn to virtualization, the enabler for TEEs that we focus on in this thesis.

2.1.2 Virtualization

Virtualization provides to software a system view of interfaces and resources that differs from the actual (physical) interfaces and resources of the system [164]. Virtualization can be used for compatibility reasons, for resource sharing, or for isolation. One of the most common forms of virtualization - *process virtualization* - is performed by operating systems to provide processes with a standard interface where context switches and variations in physical memory layout are transparent. In contrast, in *system virtualization* [164] it is the operating system itself that executes upon a virtualized interface. For example, an operating system could be hosted by another one. Alternatively, operating systems can execute side-by-side on a single physical device without another operating system hosting them. This latter form is called *type-1* [79, section 2.1] or *bare metal* virtualization. We refer to the software that provides this virtualization as a *virtual machine monitor* or *hypervisor* and to the software systems executing upon their virtualized interface as *virtual machines* or *guests.*

A type-1 hypervisor might attempt to provide its guests with a view that is identical to the physical system, modulo the extent of available resources like memory, cores, or time. This form is known as *full virtualization* and makes it transparent to the guest that it is executing in a virtualized environment. In contrast, *paravirtualization* refers to the modification of guests in order to function in the virtualized environment. Typically, such paravirtualized guests would issue system calls invoking the hypervisor for privileged functionality, so-called *hypercalls.* Alternatively, a hypervisor can be invoked on exceptions, such as interrupts, data or prefetch aborts, or a guest's attempt of executing a privileged instruction. Such *trapping* is used in both paravirtualization and full virtualization. In the latter case, the hypervisor consequently *emulates* the functionality that the guest failed to execute.

While paravirtualization can be done with the same hardware enablers that operating systems usually use, full virtualization can be hard or even impossible without additional hardware support. The minimum requirement for full virtualization is that all privileged operations trap into the hypervisor when attempted by guests. However, since guests usually maintain their own isolation hierarchy (between OS kernel and applications), a hypervisor would need to emulate hardware functionality such as system calls and memory management, if it is intended to provide the guest with this internal separation. Since that can be a tedious task, modern hardware platforms provide virtualization support such as multiple privilege rings or a 2-stage-MMU. A 2-stage-MMU allows the guest OS kernel to directly define an address translation from virtual addresses to intermediate addresses and the hypervisor to define the translation between those intermediate addresses and physical memory.

Isolation in the context of virtualization is fulfilled when guests are not able to learn about fellow-guests or influence them. Sometimes, one wants to relax this interpretation and support functionality like inter-guest communication. It can still be meaningful to speak about isolation then, given that guests cannot influence each other in any other way than via a controlled well-defined communication channel.

Separation kernels are similar to hypervisors and the two terms are often used without sharp distinction. Rushby introduced separation kernels as software that mimics a distributed environment, with guest processes executing separately from each other except for explicit communication [144]. The (security) term "separation kernel" is more common for kernels that focus on isolation and whose guest processes do not necessarily have operating systems. In contrast, the term "hypervisor" often refers to virtualization software with more virtualization functionality and complex guest systems. For instance, a hypervisor would typically apply memory translation instead of pure memory protection with a 1-to-1 mapping. Because of the overlapping terminology, this thesis often employs the term "hypervisor" for both types of system software.

2.1.3 System Aspects of Isolation

The security of a virtualized platform does not solely depend on the linear execution of hypervisor code. Actors coexisting with hypervisor threads (Figure 2.1) influence the system, among them:

- code that executes before the hypervisor, that is, boot code.
- code that executes in between two hypervisor handlers on the same core, that is, runtime firmware (e.g., for power management) or guest code. Firmware code is often unknown. Guest code can be any arbitrary code, only limited by the privilege management of the instruction set architecture (ISA).
- code on other cores. This code executes in parallel and can be another hypervisor thread, a guest, or firmware. Some of the resources accessed by a core are exclusive to that core, while other resources are shared between the cores, creating race conditions and data leakage, for instance.
- peripherals operating in parallel.
- system components that are executing on behalf of the hypervisor or the other actors above, but do so in parallel and with delays, that is, without predictable execution sequences. That is the case for co-processors, but also for the entire memory subsystem, including store buffers, caches, and memory management.

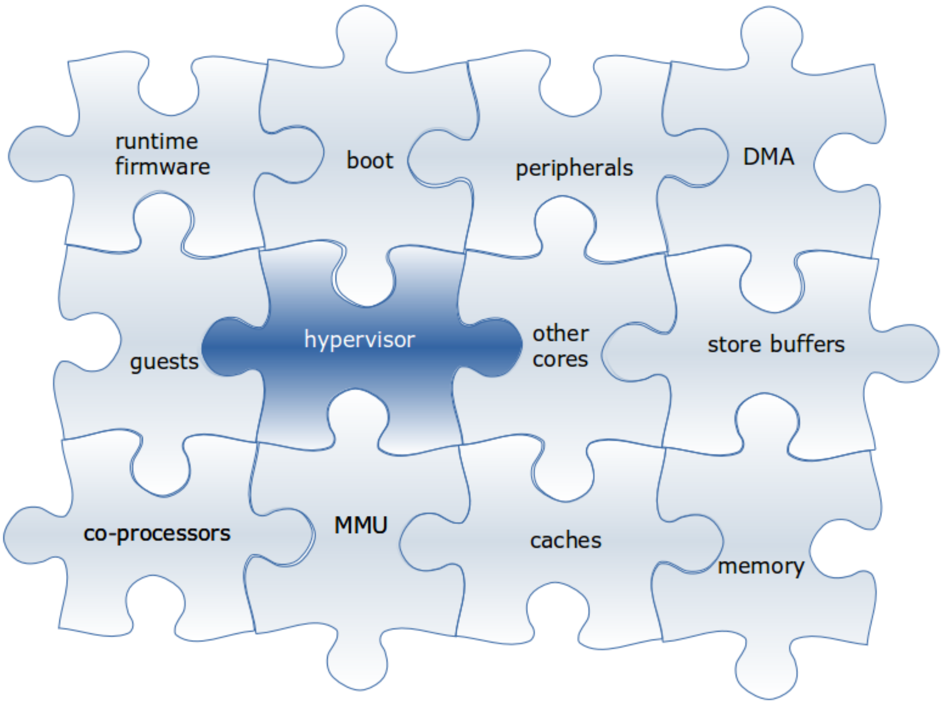

Figure 2.1: System security is influenced by many actors.

It is essential to understand how the above actors can change the system, what information they can access, and under which circumstances this happens. This knowledge allows for the correct system configuration and implementation of context switches and other hypervisor operations. However, this matter is not trivial and the past has shown that many of the coexisting actors in the system can set security at risk. This subsection surveys some of the known issues.

Much complexity is introduced through peripheral devices. On some architectures, the programmer has access to specific commands to communicate with peripherals. On other architectures like ARM, peripheral devices are mostly memory mapped, i.e., their ports are accessible via standard load and store instructions. From the programmer's perspective, memory mapped devices do not differ from actual memory. The only attribute that distinguishes them from memory is their physical address. Therefore, the MMU can be used to constrain which device ports the CPU has access to. However, some devices have *direct memory access (DMA)* and can read from or write to both physical memory and other devices on the bus. If untrusted code has access to DMA controllers, the controllers can be programmed to circumvent memory isolation. Wojtczuk demonstrates on x86 how a manipulated driver for a network interface controller (NIC) or hard disk drive (HDD) can use DMA to modify code and data of a Xen 3.x hypervisor from the (relatively privileged) dom0 guest domain [181]. DMA attacks have also been used to develop stealthy keyloggers, undetectable by the host [169]. *Input/output memory management units (IOMMUs)*, also called *system memory management units (SMMUs)* in ARM terminology, are one way to counteract DMA attacks. They can be programmed by system software to constrain the address space accessible by adjacent peripherals. However, many embedded devices do not include an IOMMU. Alternative ways of protection against DMA attacks are discussed in Chapter E. A special form of DMA attacks is based on the *FireWire* technology implemented via the Open Host Controller Interface (OHCI). As shown in [32], the interface provides DMA to external peripherals without the involvement of system software. Therefore, malicious external peripherals can compromise the system even without previously installed malware. There are countermeasures to those attacks, for instance OHCI filters, but FireWire attacks still demonstrate that threats do not only come from software.

Before the hypervisor has even a chance of setting up the machine to protect itself, firmware – such as the basic input/output system (BIOS) – and boot code are executing. We need to trust this code to load the hypervisor as expected. *Secure boot schemes* (e.g., as in [127]) use read-only memory and cryptographic means to guarantee the integrity of the hypervisor and guests. *Trusted boot schemes* also attest this integrity to the user or remote parties. These schemes can help to protect, for instance, against manipulations on the storage device that contains the hypervisor code before boot. However, we still need to rely on the correctness of the boot code sequence itself. Unfortunately, BIOS software is not standardized and rarely updated, and many BIOS systems are known to have vulnerabilities [108]. Even worse, firmware is not only a concern at boot time, but remains active afterwards. A well-known example of this is the *System Management Mode (SMM)* on Intel's x86 machines. The SMM is invoked for power management, for instance, and executes without restrictions on memory or I/O. Furthermore, it has access to the context stored by the CPU when entering SMM. It is thus quite attractive for an attacker to operate in SMM. The memory that contains the SMM handler code is supposed to be protected, but some attacks have managed to overwrite it [62]. These attacks already require at least administrator privileges, but allow the attacker to become even more powerful [62] or to install a stealthy rootkit [64, 154] that the operating system (or hypervisor) cannot detect. While SMM is specific to x86, similar issues might exist on other platforms. Also on ARM, firmware code executes during and after boot and in a powerful setting (e.g., in the Secure World of TrustZone). As pointed out in Section 2.1.1, TEEs are supposed to reduce the amount of code we need to trust. At the same time, these examples show that TEEs can reduce the ability for users to gain trust when the TEEs hide third-party code from inspection.

Side-channels are a more subtle attack vector than the ones discussed above. However, it has been repeatedly shown that they allow the extraction of cryptographic keys, for example. Furthermore, they are hard to avoid and exist in various forms. Among the most important representatives are cache channels, which can be both timing-based or purely logical [135, 143, 99, 185, 83]. Physical side-channels include the analysis of power, electromagnetic patterns, or acoustic emanations. For instance, Genkin et al. demonstrated that inexpensive equipment can extract keys over the electromagnetic channel in a few seconds from a distance of a half meter [76].

The purpose of this subsection is to give an intuition of the complexity of modern systems. There are more components that would deserve attention, for instance debug registers, but a complete survey is out of scope of this thesis. The ambition of this thesis is to improve assurance for a few pieces of the complex puzzle just described. In particular, the included results focus on the security during boot, during user mode execution, and in the context of DMA-devices.

2.2 Formal Verification

Assurance of the correctness of soft- or hardware has traditionally been derived from testing and manual inspection. However, those methods are far from complete. To increase assurance, a range of machinery has been developed to actually prove correctness. While the assurance obtained from such proofs can still vary, all approaches share the idea of creating mathematical models of the systems to analyze and subsequently derive or confirm - also mathematically - formal properties. By *formal* we mean that those properties are expressed as statements in some logic. Consequently, those methods are referred to as *formal methods* or *formal verification.*

2.2.1 Representatives of Formal Methods

Next, we survey the most prominent representatives. Note that although we attempt to categorize the different approaches, a sharp separation is not always meaningful.

Model Checking In model checking, the system to verify is modeled as state transition system and automated graph search algorithms check if the given specification holds of the system's state graph [65]. The specification is usually formulated as a temporal property, thus expressing behaviors such as "it is always (i.e., in every state) true that X" or "whenever Y occurs, eventually Z will become true". On violations of the specification, the model checking tool exhibits a counterexample. The advantages of model checking are the automation and user-friendliness. On the downside, the exhaustive exploration of the state graph is expensive (*state space explosion*). This practically prohibits detailed system models (e.g., where a machine's memory is modeled completely as a huge array of words). The model designer thus has to abstract the state, with the risk of missing relevant behavior.

SAT Solving Propositional logic formulas can be transformed into an equivalent formula in conjunctive normal form (CNF), that is, a conjunction of clauses that are in turn disjunctions of possibly negated variables [97, chapter 1.5]. Determining whether or not such a formula is satisfiable is known as the *Boolean satisfiability problem (SAT)*, which is decidable, but in general NP-complete [48]. However, in practice *SAT solvers* can decide the satisfiability of many formulas in acceptable time. Generalizing the concept of SAT solving, solvers for *satisfiability modulo theories (SMT)* (e.g., [56]) allow to include domain-specific theorems (e.g., on integer arithmetic) into the reasoning. SAT solvers can be directly applied for reasoning on state transition systems [159] or for automatically discharging sub-obligations in other proof tools.

Static Automated Program Verification Static verification of functional and safety properties for software can be automated to a large extent. The "model" is the program code itself, as source code, in an intermediate format (e.g., Java bytecode), or as binary. The specification is usually given as a contract of pre- and postconditions on code blocks. In addition to those assertions, the user can typically provide intermediate assertions such as loop invariants to facilitate the reasoning. The most prominent notation is based on Hoare-triples [93] of the form $\{P\}C\{Q\}$ expressing that if precondition P holds then program code C will progress the program to a state where Q is true. *Total correctness* additionally requires termination of C, while *partial correctness* only assures Q under the condition that C terminates. Axioms and proof rules allow to derive and combine such triples based on the operational semantics of the code language. They describe how pre- and postconditions relate in the context of for example assignments, sequential composition, or branches. Starting from the desired postcondition, those rules allow to automatically infer the weakest precondition. Alternatively, one can start from a known precondition and infer the strongest postcondition. Along the process, proof obligations are accumulated. Their conjunction - the *verification condition* - is a first order formula that can be verified by an SMT solver in many cases. Loops and less simple relations on arithmetics and data structures might require additional user interaction. Examples of static code verification tools are the Binary Analysis Platform (BAP) [43] for binary code and VCC for concurrent C [47].

Protocol Verification Outside of platform security, formal verification is for example used to verify the security of network protocols that make use of cryptographic principles. Given a model for the protocol and the attacker's behavior, they check what an attacker can achieve in the protocol. Examples of protocol verification tools are ProVerif [37, 38] and AVISPA [177].

Interactive Theorem Proving Despite the name, an *interactive theorem prover* is more of a proof checker than a proof generator, even though more and more automation finds its way into modern theorem provers. The core principle of an interactive theorem prover is that propositions can be stated in some specific logic and that inference rules allow to transform assured propositions (such as axioms) into new assured propositions. The inference rules can be both of a basic kind or derived complex rules. Among the most prominent theorem provers are HOL4 [95, 81], Isabelle [101], and Coq [33]. They all have some form of meta language that allows to operate on *terms*, i.e., expressions in the logic. For HOL4 that meta language is Standard ML (SML), a widely-used general-purpose functional language. Logical propositions in HOL4 are simply terms of Boolean type. It is important to point out that there is a difference between SML types and HOL4 types. When we say "a term of Boolean type", then we actually mean that the HOL4 type of the term is Boolean, while the SML type is `term` with `bool` as a type parameter. Proven propositions along with their premises are represented in an abstract SML-type for *theorems*. This type does not have primitive constructors. Therefore, users can only construct objects of the theorem type with sound primitive inference operations [2] [132]. SML allows to combine those inference operations to arbitrarily complex operations on theorems. Below we discuss the most common ones. *Rules* transform theorems to new theorems. For instance, the rule `SYM` would turn $1 + 2 = 3$ into $3 = 1 + 2$. *Conversions* take as input a term and turn it into a theorem that states the equivalence of that term to some other term computed by the conversion. For example, a conversion could generate the theorem $1+2 = 2+1$ from the input $1+2$. Instead of reasoning from known theorems forwards to desired theorems, the opposite is possible as well and is called *backward reasoning.* A proof goal is reduced to one or several subgoals including premises, which is then repeated recursively, generating a proof tree (or *proof stack*). This procedure is usually performed in a depth-first manner, moving upwards whenever a leaf node can be finished off completely. The proof stack is managed by a *proof manager*, that also comes with some sort of user interface when used interactively. Reductions are based on so-called *tactics*, functions from a goal[3] to a list of subgoals and a justification function. The justification function allows the deduction of the desired theorem from the theorems that would result from the subgoals. Functions that transform tactics are called *tacticals.*

Even though the most common use of HOL4 is to do backward reasoning with standard tactics, HOL4 really allows to "program" proofs more freely. Therefore, rather than understanding HOL4 reasoning as the instantiation of a fixed proof structure, it should be seen more generally as theorem processing. This allows for proof tools as demonstrated in Chapters B and C. Their development involves algorithm design and programming, just as in the development of arithmetic or graph tools. Instead of numbers and graphs, however, the basic operations are

[2]Strictly speaking, HOL4 allows to construct arbitrary theorems even without proof. However these theorems will be tagged. Tags are propagated and cannot be removed. One way of constructing such theorems is by the application of the `mk_oracle` command, see `https://hol-theorem-prover.org/kananaskis-10-helpdocs/help/Docfiles/HTML/Thm.mk_oracle_thm.html`

[3]A goal consists of a term as the proposition to prove and a list of premise terms.

inference rules operating on terms and theorems. Since the inference rules are composed from a small set of basic and sound rules, obtained theorems are sound.

2.2.2 State Transition Systems

In the following, we use the term *machine* as general concept of a computing system – be it a single processor core, a peripheral, or an entire device. A machine has components such as registers, memory, control flags, or a coprocessor, and those components can in turn have subcomponents. The *state* of a primitive component is the value it currently holds, while the state of a complex component or the entire machine is a structure comprised of the states of all subcomponents. Whether this structure is represented as tuple, record, or some other type, depends on the mathematical model of the system. In deterministic settings, machine progress can be represented by *transition functions* that take a pre-state, possibly along with some parameters, and return a post-state. If non-determinism needs to be modeled, relations between pre- and post-states are a suitable generalization. The models used in this thesis are on instruction set level and most of them do not include non-determinism, since they do not consider timing, concurrency or other behavior whose outcome is hard to predict. Chapter D is an exception and includes a model in which a CPU core executes concurrently with peripheral devices. However, we decided to reflect the unknown effect order from concurrent execution with the help of an uninterpreted oracle function. Our proofs then quantify over all possible orders. This decision allowed us to stay closer to the original CPU model than by switching to transition relations. In the following, we focus on transitions represented by functions. Transitions can be of any granularity, representing one processor instruction, only parts thereof, or entire programs. Sometimes, it is meaningful to fix the granularity. Then we use the term *step* for a transition of the chosen granularity. A *step sequence* is the chained application of steps, so that the post-state of one step is the pre-state of the next step. A chain of involved states in such step sequence is called *trace*.

The state transition systems most relevant for this thesis are the ISA models by Fox et al. [68, 69, 70]. They are available for ARM, MIPS, x86, and other architectures. The instruction sets are modelled based on official manuals and on the abstraction level of the programmer's view, thus being agnostic to internals like pipelines. The newest models are produced in the domain-specific language L3 [68] and can be exported to SML and HOL4.

2.2.3 Noninterference

In the following we introduce the property of *noninterference* as formulated by Goguen and Meseguer [77, 78] and by Rushby [145, chapter 2]. We first present the general formulation and later instantiate it to be close to the scenarios and models discussed in this thesis.

We assume a state transition system with a set S of states, some actions A (e.g., inputs or commands), and a set O of outputs. Given a pre-state and an action, the system takes a step deterministically, $\texttt{step} : S \times A \rightarrow S$. However, the free choice of the action might introduce a certain nondeterminism if no further restrictions apply. Extending the step function, $\texttt{step}^* : S \times A^* \rightarrow S$ denotes the execution of multiple steps, inductively defined over a list of actions. We assume the existence of an output function $\texttt{out} : S \times A \rightarrow O$ that takes a state and an action and returns an element of O as the output associated with that action on that state. Let D denote a set of security domains, for instance, {public, secret} or a set with one domain per process. The function $\texttt{dom} : A \rightarrow D$ associates a domain with each action. Furthermore, a security policy defines which domains may influence which other domains. We define a function $\texttt{purge} : A^* \times D \rightarrow A^*$ that takes an action sequence and a domain d and removes all actions from the sequence that belong to a domain that is not allowed to influence (i.e., that is not readable by) d.

Rushby provides an intuition for *information flow* as follows: "information can be said to flow from a domain u to a domain v exactly when actions submitted by domain u cause the behaviour of the system perceived by domain v to be different from that perceived when those actions are not present" [145, chapter 2, page 7]. In that understanding, *noninterference* is the absence of undesired information flow.

Definition 2.2.1. Let s_0 be a starting state, possibly subject to some preconditions. Then, we say that *noninterference* holds, if for all action sequences α and all actions a it is true that

$$\texttt{out}(\texttt{step}^*(s_0, \alpha), a) = \texttt{out}(\texttt{step}^*(s_0, \texttt{purge}(\alpha, \texttt{dom}(a))), a).$$

The domain of action a should not be affected by domains removed through purging.

When showing noninterference for a concrete system, reasoning on sequences of state transitions can be challenging. However, noninterference can be reduced to a set of unwinding conditions on single (pairs of) state transitions, see [145, chapter 2] and [78]. A core ingredient for this reduction are state equivalence classes defined for each domain. If two states are in an equivalence relation $\sim^d$, it should be impossible for domain d to distinguish them. It is the task of the verifier to identify a suitable relation, guided by the unwinding conditions of the following theorem [145, theorem 1]:

Theorem 1. Noninterference holds if for every domain d there is an equivalence relation $\sim^d$, such that these three conditions hold:

1. output consistency: $(d = \texttt{dom}(a)) \wedge s \sim^d t \Rightarrow \texttt{out}(s, a) = \texttt{out}(t, a)$ (for all states s and t and all actions a)
2. local policy compliance: $s \sim^d \texttt{step}(s, a)$ for all actions a whose domain $\texttt{dom}(a)$ is not allowed to influence d (for all states s)

3. step consistency: $s \sim^d t \Rightarrow \texttt{step}(s, a) \sim^d \texttt{step}(t, a)$ (for all states s and t and actions a).

Proof sketch: Consider any domain d, an original trace, its purged version, and any pair of states (one state from each trace), such that the two states are related by $\sim^d$. We induct over the original trace. For every step we distinguish two cases. Either the action of that step belongs to a domain that is allowed to influence d. Then this step is also represented in the purged version and step consistency maintains $\sim^d$. Or else the action of the step does not belong to a domain that is allowed to influence d. In that case, the step has been purged from the other trace, but local policy compliance guarantees that the relation is still maintained, when we advance only in the original trace. Finally, after any walk on a trace pair we will still be in states related by $\sim^d$. From there, output consistency allows us to conclude noninterference [145, chapter 2].

To understand what noninterference means for the scenarios and models in this thesis, we instantiate the above formalism with a system close to the systems analyzed in later chapters. We consider a virtualized single-core machine where two or more guests take turns in execution. A hypervisor handles the context switches, by storing the context of a paused guest to memory and restoring the context when reactivating the guest. For simplicity, we also assume that the hypervisor manages some flag $\texttt{flag}_g$ for each guest g indicating whether the guest's context is in the context buffer or in the registers of the processor. [4] Context switches are initiated by timer interrupts. For now, we assume that interrupts are the only way to trap to the hypervisor and that guests do not communicate. Guests do not have access to the system time. Intuitively, the established separation implies that a guest cannot learn anything about any other guest. It should not even be able to learn about other guests' existence.

We choose the set D of security domains to contain one domain for the hypervisor and one for each guest (but no further domains). Guest domains are not allowed to be influenced by any other domain. The system models in this thesis mostly represent instruction set architectures, with program code residing in memory, a program counter pointing to the next instruction, the privilege level encoded in some control register, etc. – all included in the modeled state. In such a state transition system, the concept of actions becomes almost superfluous. The state already encodes which actor (i.e., domain – guest or hypervisor) is currently active, and every pre-state only has one possible post-state. The only exceptions to this determinism are interrupt signals. We thus use actions to accommodate interrupts and to make the active domain encoded in the state more explicit for the above formalism. To that end, we choose the set A of actions to equal the set D of domains, instantiate $\texttt{dom}$ with the identity mapping, and instantiate $\texttt{step} : S \times D \to S$ with a partial function that takes a state s and a domain d and returns

[4] This information can also be concluded from the remaining machine state, for instance from the program counter and the handler code.

- the progressed machine state after one ISA step, if domain d matches the active actor in s;
- the post-state to s after an interrupt signal, if d is the hypervisor domain, but a guest is active in s.

For simplicity, this instantiation excludes interrupts fired during hypervisor phases, but accommodating them requires minor changes. Note that interrupt steps belong to the hypervisor domain. This modeling decision is motivated by purging reasons. If interrupt actions belonged to guest domains, then execution sequences purged for that guest would end up in privileged mode without any hypervisor or other handler to get the system back into user mode. This would prevent step consistency. The step function is partial, since a step on a machine can only be an interrupt or performed by the active actor/domain. There is no meaningful interpretation of a guest-action when the hypervisor or another guest is executing. A total step function that just idles for such cases would generate some traces that cannot be matched with their purged versions when attempting to show the unwinding conditions. Disallowing some action sequences by choosing a partial step function introduces the additional proof obligation of showing that purging of valid action sequences will always result in valid action sequences again. We omit detailed proofs here. The intuition is that purged sequences will only contain actions associated with one guest, and when starting from user mode with that guest active the system cannot reach a scenario in which this guest is not active any more. [5] As output function we chose a mapping that takes a state s and a domain d and returns a masked version of s, such that:

- if d matches the active actor in s, all components considered readable for d remain as in s, while all unreadable components are overwritten with standard values;
- if d does not match the active actor in s, a state with standard values only will be returned.

Since we assume that the hypervisor can access the entire system, a suitable equivalence relation for its domain is the identity relation. Conditions 1 and 3 follow trivially and so does local policy compliance, since there are no domains that are not allowed to influence (i.e., to be readable by) the hypervisor. The equivalence relations for the guests are less straightforward. Clearly, they should reflect that a guest depends on its own memory, but not on foreign memory. Thus, two states related by $\sim^g$ would be required to agree on guest g's memory, but would not restrict the states on other guests' memory. Naïvely, we would extend this pattern to other state components, such as registers, requiring two states in $\sim^g$ to agree on all guest-visible registers and being liberal on all registers invisible to g. This would fulfill

[5]Remember that we have assumed that interrupts are the only way to enter privileged execution mode.

output consistency, but leaves local policy compliance unsatisfied, since the guest-accessible registers will be used by the hypervisor and other guests when g is not active. We could relax the relation by removing the restrictions for states in which g is inactive. That is fine for output consistency because of our definition of `out`. Also, the local policy would be trivially respected in almost all cases. The downside, however, is that we would loose guarantees that we need to re-establish equivalence when the execution returns to g. To maintain these guarantees, we make use of the actual hypervisor design, which involves storing and restoring guest contexts whenever those guests turn inactive or are reactivated. We include the context buffer into the reasoning by letting it take the role of the guest-visible registers when g is inactive. Since it takes the hypervisor several instructions to store or restore the context, we make use of $\texttt{flag}_g$ to define exactly when the register data of g is considered to be found in the actual registers, (partly) in banked registers, or the context buffer of g. If context storing/restoring is correctly implemented and neither hypervisor nor other guests manipulate the context of g, then phases in which g is inactive will be transparent to g and the unwinding conditions will hold.

From the perspective of g, we can break conditions 2 and 3 further down to five different phases and their proof obligations:

- **Execution of** g: During the execution of g, the local policy is trivially respected. For step consistency, we need to show $s \sim^g t \Rightarrow \texttt{step}(s, g) \sim^g \texttt{step}(t, g)$. This property is often referred to as *non-infiltration* [90] in this thesis. When g is active, the relation $\sim^g$ relates states that agree on the observable components. Therefore, non-infiltration expresses that a variation of secret state components will not cause variations of observable components. Definition 2.2.1 of noninterference considers a composed system and an interleaving of several actors/domains. In contrast, non-infiltration can be understood as the local perspective (here of g) on noninterference. That is why in the literature – especially in the information flow analysis of programs – non-infiltration is often directly referred to as noninterference [147] or similar terms such as batch-job noninterference [19]. Chapters B, C, and D follow this convention.

  In order to show non-infiltration over the course of several instructions, we need an invariant on possible preconditions to non-infiltration. For instance, it usually has to be shown that g is not able to affect the page tables. Otherwise, it would be able to learn information in the next step and thus break non-infiltration. The integrity property that g cannot manipulate certain machine components is called *non-exfiltration* [90].

- **Change of processor mode**: In order to reason about privileged execution, several guarantees have to be established when switching the processor's privileged mode. Most importantly, the program counter must be pointing to a well-defined position in an exception handler. That is, guests should not be able to switch to a privileged execution mode and execute arbitrary code there. Similarly, endianness and instruction set flavour need to be set as expected instead of guest-defined. Interrupts need to be masked. The return address should in fact belong to g. We refer to those properties as *switching properties.*

- **Storing/restoring context**: With the help of the switching properties we can now reason about the hypervisor code. The central part here is to show that storing and restoring context complies with the local policy (condition 2). Step consistency follows trivially.
- **Independent hypervisor execution**: For the rest of the hypervisor execution, we essentially have to show that the remaining hypervisor code does not modify the context buffer or memory of guest g.
- **Execution of other guests**: Also the other guests should not modify the context or memory of g in order to guarantee local policy compliance. Furthermore, they must not perform any state modifications that would invalidate the preconditions of the properties above. To that end, non-exfiltration provides all necessary guarantees on the integrity of page tables, hypervisor data and code, and g.

In practice, the exact definition of noninterference has to be adapted to the considered system. For example, one might want to accommodate instructions that trap into a privileged handler, such as software interrupts. They differ from hardware interrupts in that they cannot be handled entirely transparent to the guest. For example, the program counter before suspension is different from the program counter after suspension. To avoid that purged executions get stuck, hypervisor steps would need to be substituted by some basic functionality of an imaginary machine that does deviate from an actual machine. The deviations would need to be considered as "secure by design".

In chapter A we extend this concept of comparing the actual execution to an imaginary machine and functionality that is secure by design. In particular, the imaginary – *ideal* – machine of chapter A is secure by design as a whole, not only in parts. This allows us to create a notion of security for settings where interference between guests actually exists, but only in the form of controlled communication.

More general notions of noninterference allow for intransitivity and/or accommodate nondeterminism [178, 125]. Intransitivity can be meaningful when a mediating actor such as a kernel is both influenced by and influencing various domains while there must not be any flow between these domains.

2.2.4 Information Flow Analysis

As discussed above, in the information flow analysis of code, noninterference is usually understood as what we also called noninfiltration. Given a set of components (or variables), we want to know possible flows of information between variables. If there are no "illegal" flows, noninterference holds. In the simplest case we divide the set of variables into *high* (secret) and *low* (public) and want to show that no information flows from high to low. In more general formulations, our domains might come from a lattice, for instance [57]. The assignment of domains to variables is called *labelling*.

Uncovering information flow requires careful analysis. In addition to *explicit* flows through direct assignments, flows can be *implicit*, that is, caused by control flow that depends on (possibly secret) values. For example, the assignment $l :=$ (`if` x `then` 1 `else` 0) always assigns a public value to l, however, in dependency on x, thus allowing an observer of l to infer the value of x. A collection of machinery has been developed by the community to both identify information flow and verify or enforce its absence. While *dynamic* approaches track flows during the execution and attempt to prevent undesired flows, *static* approaches analyse possible flows before the execution. In the latter case a labelling is either given in advance (as desired partition policy) and checked [7] or computed throughout the analysis [96].

Type Systems The most prominent class of approaches for information flow analysis is formed by type systems, as surveyed in [147]. They extend existing concepts for tracking the data type of values (e.g., integer or boolean) to tracking secrecy labels (in the simplest case high and low). To that end, the single steps of (a program's or an instruction's) semantics are matched to inference rules that gradually restrict the set of possible labels for a given value. For example, such a rule could express that any variable that is assigned a value in some conditional branch needs to be regarded at least as secret as the value of the conditional guard. This addresses implicit flows as the one discussed above, but can be too restrictive in some cases, as demonstrated by the example `if` $(h == 0)$ `then` $l := h$ `else` $l := 0$. Here, the resulting value of l does not depend on h.

Most type systems concern imperative languages. Functional languages can arguably be considered easier to analyse, since they usually have no or at least fewer side-effects compared to imperative languages. Furthermore, at least for dynamic analysis, Austin et al. [20] argue that sound tracking of value updates in dependency of branches is inherently less challenging in functional languages. [6]

2.3 Verification of Platform Security

Verification of system software (such as operating systems or hypervisors) and other aspects of platform security has been studied intensively by the community in the recent years. First, it is a field with many challenges. Kernel software involves assembly, needs to handle interrupts, and interacts directly with low-level hardware. It might even (potentially) change the premises it is running under, such as its own

[6]Consequently, [20] proposes the translation of imperative code into functional one before performing information flow analysis.

code, its address translation, operation mode etc. Second, verification of platform security solutions has a high impact. While the verification of commodity application software might still seem overly expensive in respect to the benefits, kernel software is influencing the security of the entire system.

The verification of system software is more than 35 years old [66, 130]. Surveys of Gerwin Klein [103] and Yongwang Zhao [187] give an overview of what has been achieved since then in the verification of operating systems and separation kernels. In the following, we describe a selection of projects that are most relevant to us.

Already in 1989, William Bevier verified isolation properties of a simple operating system, essentially a separation kernel for an artificial instruction set [35]. Like the separation kernel in Chapter A, Bevier's kernel schedules several processes, allows for inter-process communication (IPC), and enforces process isolation otherwise. The verification in the Boyer-Moore theorem prover [41] is performed in two steps: First, it is shown that the machine code implementation refines an abstract specification of the kernel. Second, Bevier verifies that the execution of processes upon the abstract kernel actually corresponds to the distributed non-privileged execution of the processes with dedicated communication channels, thus, that processes can only communicate as specified. The verification also covers address space isolation during user mode execution [34, section 5.3].

The most prominent system software verification project targeted the seL4 microkernel [104, 105]. The first of several steps was the verification of functional correctness through an Isabelle refinement proof that related an abstract specification with an intermediate specification and that one in turn with the C level source code. In addition, code safety properties like the absence of null pointer dereferences were shown. This first step took several person years to complete, but the established refinement allowed to verify further properties on a high level and transfer them to the implementation level with lower effort. Consequently, in later publications the seL4 verification team showed security properties such as integrity [157] or noninterference [124, 125] and extended the verification down to ARM binary level [156].

The Verisoft XT project deals with the pervasive formal verification of computer systems and is comprised of several sub-projects. One of them investigates the application of Microsoft's verification tool VCC in the verification of Microsoft's hypervisor Hyper-V. VCC allows the automated verification of contracts for C handler code. In order to include the underlying hardware and guest execution into the reasoning, an emulator for the hardware (in [5] a simplified MIPS machine) was written in C [5, 134]. The targeted main property constitutes that every state which a guest can reach can also be reached when the guest executes on an isolated machine. The property was shown for a simplified hypervisor. Another subprojects of the Verisoft XT project verified the PowerPC version of the microkernel-based partitioning hypervisor PikeOS [27]. They employed VCC again to show the functional correctness of system calls, including (inline) assembly blocks. To that end, they captured hardware components in the ghost state.

Heitmeyer et al. and their verification of a separation kernel [90, 91] influenced terminology and understanding of separation properties in this thesis. For their topl-level specification (TLS) of separation kernel and guests (partitions) they verify the following properties:

- No-exfiltration: A partition cannot directly manipulate other partitions.
- No-infiltration: A partition is not influenced by data outside the partition.
- Temporal separation: When a partition is not processing data, its memory is clear.
- Separation of control: The data memory of inactive partitions does not change.
- Kernel integrity: A partition cannot directly manipulate kernel memory.

In this thesis, we merge kernel integrity and no-exfiltration to non-exfiltration. Heitmeyer et al. use the PVS theorem prover to show that the listed properties hold for the TLS. Furthermore, they prove contracts (in terms of pre and post conditions) on the 3000 C/assembly lines of kernel code and show that the code satisfies the TLS. Heitmeyer et al. include I/O buffers into their model, but since their buffers are rather abstract, they do not cover all particularities of peripheral devices (cf. Chapter D). Furthermore, these buffers only model external communication. Channels for inter-guest communication as discussed in Chapter A lead to the additional requirement of dealing with mutual influences between guests.

Costanzo et al. [49] report on the Coq verification of noninterference for the mCertiKOS kernel, which is implemented in a combination of C and assembly. With abstraction/simulation techniques and the help of a verified compiler they guarantee security on binary level. Inter-process communication is disabled in the verified kernel version.

Also model checking can give meaningful results in hypervisor verification, as, for instance, demonstrated for SecVisor [155, 71]. To that end, the system was modeled both with and without attacker. It was then checked whether there is a reachable state of the compound model where the two sub-models deviate in central data structures. The approach allowed to identify two vulnerabilities on design level. In [175] the model checker CBMC was used to verify memory integrity and other properties about the hypervisor framework XMHF for x86 with virtualization support and DMA devices. The verification of XMHF was recently revisited as case-study in [176]. The paper describes überSpark, an architecture for automated compositional verification of security properties of hypervisors. Employing static analysis, the architecture can deal with hypervisor extensions written in C and assembly, without the need to reverify the entire hypervisor with every extension.

The Muen separation kernel [44] was written in SPARK, which comes with built-in support for static analysis. SPARK tools allowed to verify the absence of runtime errors and exceptions, such as overflows and divisions by zero.

In most kernel verification projects the proofs influence the kernel design or specifications are chosen to suit a particular kernel. In contrast, the feasibility study presented in [150] reports on the verification of a third-party kernel implementation against an independent general reference specification. In particular, the authors discuss an Isabelle/HOL refinement proof for the open-source XtratuM kernel with more than 10k lines of C code. They employ the C-parser tool developed by NICTA to translate C code into an Isabelle/HOL representation. The paper described work in progress. In fact, the work is still ongoing as of today, but the authors extended their focus to multicore settings, involving non-trivial additional challenges.

In some projects, formal verification supported Common Criteria (CC) evaluation on high assurance levels, for instance on evaluation assurance level (EAL) 6+ for the INTEGRITY-178B real time operating system [142]. INTEGRITY-178B serves as separation kernel and provides fault containment, timing guarantees, and isolation. The central security property of [142] generalizes the "GWV" non-infiltration theorem in [180] to account for dynamic scheduling. Also the work of Heitmeyer et al. [90] was carried out to support CC evaluation, in that case even for EAL 7.

2.3.1 The Projects PROSPER and HASPOC

The PROSPER project [139] is a collaboration between KTH Royal Institute of Technology and SICS Swedish ICT. Started in 2011, it is a 5-year project founded by the Swedish Foundation for Strategic Research (Stiftelsen för Strategisk Forskning - SSF). The goal is to apply type-1 virtualization to develop isolation solutions for widely used embedded platforms (Section 2.3.2) and to formally verify those. To that end, the project developed a single-core hypervisor for ARMv5 and ARMv7 processors. Among the supported platforms are Beaglebone, Beagleboard, Beagleboard-xM, NovaThor, the Integrator development board, and emulated platforms within the OVP framework or upon Qemu. We apply paravirtualization and have so far adapted FreeRTOS, Linux 2.6, and Linux 3.10. Besides its focus on security and a thin code base, one of the hypervisor's main features is the provision of secured communication channels between guests. In a first step towards the hypervisor verification, we verified isolation properties for a simplified version, described in Chapter A. Instead of vanilla noninterference this work actually takes inter-guest communication into account. The verification was performed on binary level [54]. In order to support Linux guests, virtual guest modes (kernel, user) for intra-guest-separation were introduced and guests were enabled to manage their memory mappings dynamically. Newer features include network support with DMA protection and monitor functionality to prevent guests from executing untrusted code. The hypervisor is available as open source [162].

HASPOC (High Assurance Security Products On COTS platforms) [87, 127, 39] is a 2-year spin-off project of PROSPER and funded by Vinnova's Challenge Driven Innovation program. Its original intention was to mature the PROSPER hypervisor for commercial use together with industrial partners. However, early in the project we recognized the need to shift to the new ARMv8 architecture, which now allows us to provide fully virtualized execution (modulo inter-guest communication). Further new features include multicore support, a secure boot, and the preparation for a future CC evaluation.

2.3.2 Embedded Systems and ARM

Both PROSPER and HASPOC focus on embedded systems. While this term is overloaded to some extent, our prototypes target various ARM-systems. In a strict interpretation, an *embedded system* is a computing system that is part of – *embedded in* – a larger system and provides some dedicated functionality there, thus differing from a general purpose computer especially in its minimal end-user programmability [88, chapter 1]. Typical examples are control units in vehicles, washing machines, or medical equipment. However, in everyday use of the term "embedded systems", the criteria tend to be less sharp. For example, an entire wireless sensor node is usually regarded as embedded device, even though it is more of a complete system in its own right - with the sensor and the radio unit as I/O peripherals - rather than part of a larger system. Even smartphones or tablets are sometimes considered embedded systems, despite being programmable general purpose computing devices. Several circumstances support this view: first, historically mobile phones were indeed special purpose devices. Second, even today mobiles and tablets still are to some extent limited in resources and functionality when compared to personal computers. And finally, their architecture is in some respects closer to (other) embedded systems than to PCs.

One important architectural property that smartphones and tablets have inherited from embedded systems is the principle of deploying the majority of the system components on one single chip, the so-called *System-on-a-Chip (SoC)*. While the processor, the main memory, the graphics processing unit (GPU), etc. in a PC are separate building blocks plugged into the motherboard, they are all manufactured together in SoC designs. This practice also influences the division of the market. The designer of the most dominant processors for embedded devices including smartphones and tables, the *ARM Holdings PLC*, does not manufacture processors themselves. Instead, they license their design, so that other companies can manufacture ARM processors on manufacturer-specific SoCs. ARM processors implement *reduced instruction set computing (RISC)*. RISC instruction sets are small and contain only the most essential instructions. On the one hand, this implies that implementing a certain task often requires more instructions on RISC machines compared to machines with a more complex instruction set. On the other hand, RISC instructions require less cycles on average, which benefits performance.

ARM instructions are encoded rather uniformly, following a common pattern throughout all instructions of an instruction set. However, ARM processors implement several instruction sets. Besides the standard 32-bit-ISA, Thumb, ThumbEE, and Jazelle also include 8-bit and 16-bit instructions. The newest ARM processors are based on a 64-bit architecture and find their way even into traditional server systems [52].

Several virtualization solutions exist for embedded systems, in particular for mobile devices (see [160] for a comprehensive survey).

2.3.3 Verifying System Aspects of Isolation

Many system software verification projects focus on the verification of handler code. The hardware environment that the system software executes in – memory management, peripherals, etc. – and the abilities of unprivileged code are often ignored or only included axiomatically. However, as we have seen in Section 2.1.3, the environment in which an operating system kernel or hypervisor executes in, is critical for the overall system security. Moreover, with more virtualization support, hardware is doing more of the actual isolation work, while software remains mostly responsible for the correct configuration of that hardware. Formal properties of the system software's environment are thus becoming *the* properties that we are actually interested in for formal verification. This subsection provides a short overview of relevant work and open challenges.

Hardware Security Kernels Security kernels are not always implemented in software. The partitioning system of the AAMP7G microprocessor can be seen as a separation kernel in hardware. Wilding et al. describe in [180] how exfiltration, infiltration and mediation theorems for AAMP7G were shown with the help of refinement proofs.

Azevedo de Amorim et al. describe a processor that operates on data tagged with security labels [22]. Using refinement techniques, they show noninterference in Coq.

Memory Management While the hardware security kernels of [180] and [22] are interesting contributions towards new and secure processor architectures, it is still likely that the commodity architectures widely used today will maintain their dominance for quite some time. When verifying isolation for such commodity systems, a central aspect is how software and hardware collaborate to establish isolation. Memory management is one of the main elements in that context. Daum et al. [55] incorporate memory management into seL4's formalization. This extension allows them to distinguish individual user processes and to restrict the processes' memory accesses to their respective virtual memory. Barthe et al. [25] show isolation properties for an abstract Xen-like hypervisor modelled in Coq, with focus on memory management. Bolignano et al. [40] reason about the management of shadow page tables by a paravirualization-based hypervisor. Through abstraction/refinement methods they are able to verify memory isolation on implementation level. Nemati et al. [129] verify isolation for a hypervisor that employs direct paging. Several efforts were undertaken to include caches into the reasoning [26], including translation lookaside buffers (TLBs) [3].

Peripherals Peripherals were formalized in, for instance, [92, 4], who present models for a memory mapped HDD. The main goal is to be able to reason about the actual programming and behaviour of the storage device, in order to verify page fault handling [6]. Security concerns as they arise from DMA are not covered. However, the authors deal with another important issue, namely the concurrency of CPU and peripherals. In order to sequentialize the execution, they introduce an oracle over the effect order and quantify over all possible instantiations of that oracle. Finally, they prove reordering lemmas to minimize the otherwise high number of possible execution orders to consider.

Duan et al. developed general device model frameworks [61, 60] and integrated them into several versions of the ARM ISA models from Cambridge. Using their frameworks, they verified functionality, safety, and timing properties for a UART device and its driver. Again, DMA is not considered.

An interruptible OS kernel dealing with peripheral device drivers is the subject of [46]. The Coq verification focuses on interrupts and on functional correctness of device drivers. Still, the authors report that parts of their proofs concern the isolation between DMA devices and non-device related kernel components. Because of this isolation they can group a peripheral device, its driver, and their shared memory into an abstract object. However, we could not find details on how the direct memory access is modelled and how exactly the DMA related isolation and abstraction proofs are achieved.

Monniaux modelled a USB controller and employed the Astrée static analyzer [50] to show that controller and driver transfer data correctly [122]. The reasoning covers asynchronous DMA. Isolation from untrusted software is not discussed.

The proofs for seL4 assume that DMA is disabled [104], but the project is currently working on a verified version of the kernel with SMMU. [7] Hypervisor verification relying on IOMMUs is described in, for example, [175] and [176].

What is missing is the mediation of DMA for untrusted virtualization guests and the verification of isolation in such a setting when no IOMMU is available. These challenges are addressed in Chapters D and E.

Unprivileged Software Not only hardware, also user mode processes can break isolation, either due to a vulnerability in the ISA or because of insufficient mediation by privileged system software handlers. Kernel verification usually includes the possible kernel-user interactions through system calls, interrupts, and exceptions (e.g., in [142]). However, this is just a part of the influence that user processes have. It is also essential to understand which system components unprivileged code can access while the kernel is not active. Sometimes, such influence of user processes is axiomatized. But it is seldom verified based on actual ISA models. As acknowledged by [49], there is a need for comprehensive models and the thorough analysis of arbitrary user-mode assembly execution. Such a task is not trivial. Since user processes are not constrained to specific known code, we need to analyze all possible

[7] `https://wiki.sel4.systems/FrequentlyAskedQuestions#What_about_DMA.3F`

instructions users can execute, to understand how information flows through the machine. This can be tedious, given that specifications such as the ARM Architecture Reference Manual (ARM ARM) [13] comprise thousands of pages of text and pseudo-code. Furthermore, processor designers regularly release extensions and corrections to their architectures [140], requiring continuous re-evaluations. A high degree of automation is needed to handle both ISA size and evaluation frequency. For a long time, information flow analysis of the ISA has not received much attention by the scientific community. But recent work indicates that this is about to change. Sinha et al. [163] formalize a simple version of the x86 instruction set with SGX extensions. Among other purposes, they use the model to show that non-SGX code cannot influence the TEE by any instruction other than a write to input memory. Very recently, ARM published information on their shift to automatically generated, machine-readable, and executable ISA specifications [140]. As an exercise, they used their model to test noninterference properties of the ARMv8-M security extensions. To that end, tests on critical procedures iterated over all possible configurations/modes and scanned automatically generated dynamic dataflow graphs for information leaks. This testing identified potential security attacks. While both examples demonstrate the relevance of ISA information flow analysis, we are not aware of any such analysis on entire commodity ISAs other than our own work presented in Chapters B, C, and D. Parallel to our work, Andreescu et al. worked on machinery that does not address ISA analysis explicitly, but that could be applied to support that goal. In [9] they present an approach to automatically infer, from functional state transformation code, which state components remain unchanged throughout the transformation. Such machinery could be applied to infer non-exfiltration properties of, e.g., functional ISA specifications. Also, knowledge on which parts of the state remain unchanged can accelerate preprocessing for tracking state changes.

2.3.4 Thoughts on Validity

With processor specifications comprising thousands of pages, it is not self-evident that processor models reflect the physical hardware in a sufficiently trustworthy manner. To address this issue, Fox and Myreen validate execution results of random test instructions on a physical ARM machine against their ARM models [70]. Other validation methods include – for instance – the execution and comparison of entire programs on both the model and real hardware. ARM admits the need for trustworthy specifications and declares this need as one of the motivations behind the shift to their own machine-readable and executable specifications [140]. This step allows ARM to unify all their different specification forms, or as Alastair Reid expresses it: "Although ARM publishes an official specification, the full requirements are really distributed around many different places in the company: the AVS suite, the reference simulator ARM uses for processor verification, and the processor implementations. The act of testing all these different instantiations of the specification against each other has the effect of centralizing this specification in a single location" [140, Section IV.E]. One of their validation efforts manifests itself in the verification of their lower-level processor models (including pipelines, for instance) against ISA specifications by bounded model checking [140, 141].

As Ken Thompson discusses in his Turing Award lecture [172], trust in software can resemble a chicken and egg game. Even if we can inspect and verify the source code of a program, we do not know if the compiler would add a back door to the program. We might have the ability of inspecting the source code of the compiler in turn, but if the compiler that compiled the compiler has backdoors, there is no guarantee for a secure end-result either. One might think that this chain comes to an end at some point. However, there are compilers that are used to compile their own source code and Thompson demonstrates that it is possible to "teach" them malicious behavior that they and their descendants in the chain will maintain even if the "DNA" in the form of source code will be cleaned again. Diverse Double Compiling [179] allows to test one compiler with another in respect to such hidden behavior, even if the compilers differ in attributes such as target platforms or performance. If we get hold of a number of compilers for the same language and assume that at least one of them is trustworthy, then Diverse Double Compiling will allow to uncover any contradiction, even if we do not know which of the compilers is trustworthy. Still, establishing trust in compilers is not easy and the problem does not end there. Assemblers and other infrastructure are also affected [179]. A possible conclusion to draw is that verification on lower levels might help to reduce dependencies and the TCB. At the same time, hardware can contain backdoors, as well. Furthermore, compared to software products, hardware specifications are usually harder to obtain, to inspect, and to compare with the end product. But even if all sources are available in a hardware description language, if the machinery that synthesizes these sources is malicious, stealthy backdoors can be introduced again.

New developments in platform security enable backdoors that are not even detectable by inspecting the final hardware product. If Intel's private SGX keys are leaked, SGX enclaves can be compromised without leaving a trace. [8]

[8] Joanna Rutkowska, Invisible Things Lab, in the post "Thoughts on Intel's upcoming Software Guard Extensions (Part 2)" in the blog of the Invisible Things Lab, September 23, 2013, `http://theinvisiblethings.blogspot.se/2013/09/thoughts-on-intels-upcoming-software.html`

Chapter 3

Contributions

The goal of this thesis is to strengthen the assurance in the architectural environment in which system software such as a hypervisor runs. This is achieved by both the design of new solutions and the analysis/verification of platform and system aspects. While not all open challenges (see Sections 2.1.3 and 2.3.3) can be addressed here, we contribute to some pieces of the puzzle:

- **User-mode execution** (papers A, B, C): Which guarantees can we give while user processes are executing? Which registers can they manipulate? Which CPU components can they learn from? What can we assume once we are back in privileged mode? How do we extract this information from ISA models in the best way?
- **DMA-peripherals** (papers D, E): How can we maintain those guarantees when DMA peripherals are present? How do they need to be programmed? Can DMA controller sharing be done in a secure way?
- **Boot** (paper F): How can we maintain security if we do not wish to use the isolation services of a hypervisor all the time?

Table 3.1 gives an overview on how the different papers relate to those and other areas (hypervisor design, ISA verification, peripherals, boot, performance & usability aspects, overall hypervisor isolation), as well as on which papers focus more on verification and which more on the development/design of security solutions.

Readers Guide Readers that want to approach the included papers selectively are encouraged to start with Chapter A. This paper provides an overview of our approach to the verification of hypervisors or separation kernels. It sets the scene for papers B, C, and D. All three discuss how to obtain guarantees on user mode execution, a property that is included as a lemma in paper A. Since these three papers build upon each other, C is suggested to readers with limited time, because it describes the most general and matured work. For people that are more interested

paper	content area						contribution type	
	hypervisor	ISA	peripherals	boot	perf.	overall	verification	design
A	X	X				X	X	
B		X					X	
C		X					X	
D		X	X				X	
E	(X)		X				(X)	X
F	(X)			X	X			X

Table 3.1: Relation between papers and contributions

in the actual design of hypervisor and SoC functionality, Chapters E and F are recommended. Furthermore, Chapter E provides some background on the principles applied in early versions of the PROSPER hypervisor, including memory access control with ARM domains.

3.1 Summary of Included Papers

This thesis is comprised of six papers, originally published in peer-reviewed conference and workshop proceedings. In this section we first summarize the content of the papers and relate them to the overall thesis. Then we give account of the contributions the author of this thesis has made to them. Formatting and style are unified. Furthermore, the bibliography is shared between all papers and the first part of the thesis. Minor details in Chapter A concerning formulations have been changed from the original paper. Chapters C and F contain additional figures compared to the original publications. Also, the discussion on unpredictable behaviour in Chapter C has been slightly extended. Chapter D extends the original paper with a discussion on weak memory models. When not stated otherwise, paper contents are included as originally published.

Paper A: Formal Verification of Information Flow Security for a Simple ARM-based Separation Kernel

originally published as *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 ACM SIGSAC Conference on Computer & Communications Security (CCS), November 2013, pp. 223-234.*

Content The PROSPER project (see Section 2.3.1) started with the verification of a simple separation kernel. Even though this separation kernel only supported two guests with a static 1-to-1 address mapping, the work established some important results: instead of vanilla noninterference we showed isolation modulo explicit communication. To that end, we exhibited a bisimulation between a detailed model of the virtualized system and its top-level specification in the form of a distributed system. This bisimulation was broken down into several phases, including handler code execution, guest execution, and mode switching. We discussed how to verify the single parts and how to combine them to the overall security property. Handler code is verified on binary level, employing BAP. The other parts of the proof are performed in HOL4.

Individual Contributions This paper was a group effort from all the authors. Together with Narges Khakpour I was responsible for the verification of security properties for user-mode execution, more detailed in paper B. Moreover, I actively participated in the discussions of the overall approach, contributed to the text, and reviewed other authors' sections.

For this Thesis Minor details in Section A.1 concerning formulations have been changed from the original paper. Otherwise the content remains as originally published.

Paper B: Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties

originally published as *Narges Khakpour, Oliver Schwarz, and Mads Dam: "Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties", in Springer International Conference on Certified Programs and Proofs (CPP), December 2013, pp. 276-291.*

Content An essential part of the bisimulation proof described in paper A is the guest execution phase. In particular, we proved non-infiltration, non-exfiltration, and mode switching properties (see Section 2.2.3) for user mode execution of the ARMv7 ISA. Paper B describes the details of this work, including extensions of the ISA model to accommodate memory protection, exact specifications, and our proof machinery. As for the latter, we applied proof rules for compositional reasoning. This allowed for automation to a large extent, driven by an SML tool that we developed.

Individual Contributions I am the author of the early versions of property specifications and proof rules, including the verification of the latter. Later versions were jointly driven by Narges Khakpour and myself. Narges is the author of the SML-tool that applied the proof rules to automate the analysis. I applied her tool to most of the ARMv7-instructions, which required me to handle some special cases manually (or semi-automatically). While I focused on instructions that do not switch privilege mode, Narges analyzed the instructions that switch from user mode to some privileged mode. The text of the paper was mainly written by Narges and me in equal shares, with some contributions by Mads Dam, who also helped with discussions during the verification phase.

Paper C: Automatic Derivation of Platform Noninterference Properties

originally published as *Oliver Schwarz and Mads Dam: "Automatic Derivation of Platform Noninterference Properties", in Springer Software Engineering and Formal Methods (SEFM), July 2016, pp. 27-44.*

Content Paper B was the first analysis of ISA isolation properties, namely for ARMv7. With the initiation of the HASPOC project (see Section 2.3.1) and the shift to ARMv8, we realized that it should be possible to automate the analysis process much more than in paper B and to produce a largely automated tool, easily adaptable to other platforms. The result is described in paper C. First, the new approach does not require the user to input a candidate partitioning of system components into user-accessible and user-inaccessible components. Instead, it finds this partitioning automatically, while staying both sound and accurate. Second, the need for manual proofs was reduced. The compositional approach of paper B analyzed subprocedures isolated from the context they occur in and thus occasionally missed important constraints to guarantee security. The new approach avoids this issue by analyzing entire instructions as a whole. This decision requires more sophisticated preprocessing and proof tactics, especially to handle complexity.

Individual Contributions I performed the entire work by myself, but with continuous comments by Mads Dam, both throughout the actual work and during the writing process.

For this Thesis Figure C.1 has been added. Also, the discussion on unpredictable behaviour (Section C.7) has been slightly extended. Otherwise the content remains as originally published.

Paper D: Formal Verification of Secure User Mode Device Execution with DMA

originally published as *Oliver Schwarz and Mads Dam: "Formal Verification of Secure User Mode Device Execution with DMA", in Springer Haifa Verification Conference (HVC), November 2014, pp. 236-251.*

Content Paper D extends paper B with a framework for peripheral devices with DMA. The framework is general and agnostic to concrete device behavior. The proven security properties of the overall composed system can then be reduced to contracts for concrete devices, as demonstrated in [84]. One of the challenges is to accommodate the concurrent execution of peripherals and CPU. Even during a single instruction of the core, a device can access the memory of the system several times. It is essential to cover a fine granularity of interactions between the model components.

Individual Contributions I extended the proof scripts of paper B with a general device framework and repeated the proofs for the new interleaved setup. Mads Dam contributed some text and comments to the paper, but the majority of the text was produced by me.

For this Thesis Chapter D extends the original paper with a discussion on weak memory models (Section D.8). This discussion substitutes the original justification of an assumption we made on the atomicity of memory read requests sent by peripheral devices.

Paper E: Securing DMA through Virtualization

originally published as *Oliver Schwarz and Christian Gehrmann, "Securing DMA through Virtualization", in IEEE Complexity in Engineering (COMPENG), June 2012, pp. 1-6.*

Content As discussed in Section 2.1.3, DMA can be a threat to memory isolation. While modern desktop and server processors feature IOMMUs to counteract, many embedded systems – especially low-end systems – do not include such protection. In paper E we described how a hypervisor can maintain isolation despite DMA by filtering access to the direct memory access controller (DMAC). To that end, the hypervisor traps all access attempts to the DMAC and programs the DMAC only after checking the address range of the DMA request. We showed that this programming should be done in an atomic manner and thus argued for the exploitation of shadow data structures. The paper includes benchmarks and a very simple formal Coq proof on the design of the approach.

Individual Contributions I did the major work presented in the paper, including the detailed design of the solution, its implementation, benchmarking, and verification. Most of the paper's text was written by me, but the publication also includes contributions from Christian Gehrmann, who furthermore had the idea to work on the topic. Mads Dam helped with the verification section and language review.

Paper F: Affordable Separation on Embedded Platforms

originally published as *Oliver Schwarz, Christian Gehrmann, and Viktor Do: "Affordable Separation on Embedded Platforms: Soft Reboot Enabled Virtualization on a Dual Mode System", in Springer Trust and Trustworthy Computing (TRUST), June 2014, pp. 37-54.*

Content While type-1 virtualization serves as an enabler for TEEs, one of its disadvantages is its performance overhead, especially on platforms without virtualization support. This overhead might be justified when the achieved isolation is frequently needed, but not necessarily when a hypervisor controls the system in long periods without the need for isolation services. Paper F describes a boot procedure that allows to securely turn virtualization on and off during runtime, without the need to perform a complete reboot. The approach only requires a few minor additions to the SoC.

Individual Contributions The paper is based on a patent held by Christian Gehrmann. I implemented the suggested platform extension in the OVP emulator and the actual boot code. Moreover, I performed the literature review, took benchmarks together with Viktor Do, and lead the writing of the paper, which was almost entirely written by me. Viktor contributed with the required modifications in hypervisor and guests. He also commented on the text and produced some of the figures in the paper. Christian contributed with comments throughout the overall process and with some writing. In particular, he provided valuable input to the performance discussion.

For this Thesis The flowchart in Figure F.3 has been added. Otherwise the content remains as originally published.

3.2 Further Publications

Besides the included papers listed above, the PhD candidate is also co-author of the following work, not part of this thesis:

- *Rolf Blom and Oliver Schwarz: "High Assurance Security Products on COTS Platforms", in ERCIM News (102), ISSN 0926-4981, pp. 39-40.*
- *Mats Näslund, Christian Gehrmann, Christoph Baumann, Hans Thorsen, and Oliver Schwarz: "A High Assurance Virtualization Platform for ARMv8", in Proceedings of the European Conference on Networks and Communications (EuCNC) 2016.*

Chapter 4

Conclusions

In the background part of the thesis we have discussed the importance of isolation for platform security, that it is essential to keep the TCB of such isolation small, and that hypervisors are one option for implementing theses goals. A small TCB allows for high assurance, in particular through formal verification. We have discussed several projects that contributed to the verification of hypervisors and other system software. But we have also seen that the overall security does not depend on the hypervisor alone, but also on system aspects, that is, the environment of the hypervisor. The papers included in this thesis target some of those system aspects: (i) the instruction set architecture and the system components it makes available to guests, (ii) peripherals, in particular those with DMA, and (iii) boot procedures, combining security with performance.

As for ISA analysis, we were the first to publish the verification of isolation properties of a commodity ISA. At this point, we provide machinery to perform this analysis in an automated manner. In the future, we plan to improve robustness and performance. Furthermore, the tool of paper C should be extended to cover more properties such as non-exfiltration or mode switch properties. One of the future challenges will be to support models of different sources (e.g., of [140]), preferably also on lower levels (e.g., featuring pipelines).

The protection of memory isolation from peripherals with DMA capabilities is covered in papers D and E. While paper E discusses how to guarantee that DMA controllers used by several guests are configured in a secure way, paper D allows to conclude overall platform isolation properties based on such secure configuration. Both publications focus on systems without IOMMU. However, since IOMMUs become more common, future work should include them into the verification work.

This thesis addresses just small puzzle pieces of the overall goal to improve assurance regarding the system aspects that low level code such as hypervisors depend on. Other aspects are left for future work. Among them are the several processor extensions that hardware designers release and that conflict with the simplified system view frequently applied in the verification community. One of the future challenges is the inclusion of trusted computing into the verification of isolation solutions.

Part II

Included Papers

Paper A

Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel

Mads Dam, Roberto Guanciale, Narges Khakpour,
Hamed Nemati, and Oliver Schwarz

Abstract

A separation kernel simulates a distributed environment using a single physical machine by executing partitions in isolation and appropriately controlling communication among them. We present a formal verification of information flow security for a simple separation kernel for ARMv7. Previous work on information flow kernel security leaves communication to be handled by model-external means, and cannot be used to draw conclusions when there is explicit interaction between partitions. We propose a different approach where communication between partitions is made explicit and the information flow is analyzed in the presence of such a channel. Limiting the kernel functionality as much as meaningfully possible, we accomplish a detailed analysis and verification of the system, proving its correctness at the level of the ARMv7 assembly. As a sanity check we show how the security condition is reduced to noninterference in the special case where no communication takes place. The verification is done in HOL4 taking the Cambridge model of ARM as basis, transferring verification tasks on the actual assembly code to an adaptation of the BAP binary analysis tool developed at CMU.

A.1 Introduction

The design of secure systems needs to ensure that software components belonging to different security domains are adequately isolated from each other, such that only authorized communication can take place between them. One way of achieving this is by dedicated hardware, e.g. TPM's or SIM's. This, however, carries significant overhead, in terms of the hardware itself, and the associated infrastructure. An alternative is to execute the components in isolated partitions on shared hardware, using low-level software execution platforms such as separation kernels [144, 90] or secure hypervisors [75, 149, 117]. A key requirement of this solution is the verification of the tamper resistant trusted computing base, preferably by use of formal methods. Significant progress has been made recently in this direction, cf. the seL4 project [105], Microsoft's Hyper-V project [111], and Green Hills' CC certified INTEGRITY-178B separation kernel [142].

Our focusing scenario consists of an "untrusted" component (e.g. a smartphone software stack) that interacts with a set of trusted services, such as a virtual SIM card, or a virtual TPM. A shared execution platform for this scenario needs to provide the following minimal functionality:

1. Isolation of component resources
2. A communication mechanism that plays the role of external communication lines in a physically distributed system
3. A scheduling mechanism to commit shared resources (e.g. processors) among the components.

In this paper we present a proof-of-concept design and machine level verification of the PROSPER separation kernel for ARMv7 [13], which supports the above functionality. The kernel allows the execution of two component systems, such as a smartphone OS with a virtualized SIM application, on top of a single physical machine. The interesting feature of this set-up is that explicit, kernel-supported communication between partitions is essential, and a critical aspect of the security analysis is to ensure that this communication does not introduce (deliberate or accidental) side channels that can be exploited by an attacker.

This security analysis is far from trivial. The objective of a separation kernel[1], following Rushby [144], is first to make it appear that each component system is executed on a separate, isolated, machine, and second to ensure that communication can only flow as authorized along known external channels. However, there are several problems in delegating communication to an external agent. First, it entails an extension of the trusted computing base to include the external channel itself. Second, virtualizing the component systems without also virtualizing the channel connecting them is hardly a reasonable design. Third, potential side channels are ignored. In a case such as this, where a partition must be able to access the virtualized SIM application at will, communication can convey critical timing information that an attacker can exploit to extract key material, as is well known [106].

[1]We use the label "separation kernel" in this paper mainly since no support for user/kernel space virtualization is provided to the component systems, but the borderline between separation kernels and secure hypervisors, and our use of the associated terminology, is admittedly fuzzy.

We propose instead an approach where communication between partitions is made explicit in the top level specification (TLS), and information flow is analyzed in the presence of such an intended communication channel. We formulate the TLS such that it directly formalizes, in sufficient detail, the set of computation paths both allowed and required at the implementation level, and then we check that the implementation indeed satisfies this specification. The question is how to do this, if it can be done while maintaining a satisfactory account of isolation, and if it can be done at a satisfactory level of abstraction (such that the TLS does not conflate to become identical to the implementation itself). In this paper we present a proof-of-concept solution in the sense that functionality is limited as much as meaningfully possible, but such that the specification, implementation and correctness proof is carried through in complete detail from TLS to realization for ARMv7, and proved correct at the instruction reference semantics level using the HOL4 model of ARM developed at Cambridge [70].

In our case, the goal of verification is to show that there is no way for the partitions to affect each other, directly or indirectly, *except* through the intended channel. In particular, there should be no way for a partition to access the memory or register contents, by reading or writing, of the other partition, other than when the communication is realized by explicit usage of the intended channel, by both partitions in collaboration. This is not an easy property to reconcile with the standard information flow tools such as noninterference (NI) or intransitive noninterference:

- NI is problematic since the very purpose of the kernel is to *allow* rather than prevent information flow (through the intended channel). For the sake of illustration consider the recent NI-based verification of the seL4 kernel [125, 124]: A critical step in that work boils down to a proof of the *absence* of information flow from the previously scheduled partition to the scheduler itself, in order to prevent the scheduler being used as a communication channel. This type of approach is not applicable in our setting since communication must be allowed to affect the partitions in ways that are outside our control, as the content and behaviour of the partitions are not statically known. Moreover, for the same reason we have no control over what, where, or how the channel is intended to be used, and thus the various NI-based declassification schemes (cf. [148]) do not help.

- Intransitive noninterference [145] relaxes NI with the possibility to add unknown, but trusted, intermediary agents through which information flows can be required to pass. In our case that agent is the kernel, which is known, and not a priori trusted. This makes intransitive noninterference difficult to make use of in the present context.

We thus take a different approach. We formulate the TLS as an "ideal model" which satisfies the required separation properties by construction, and then reduce correctness to trace equivalence w.r.t. a "real model", reflecting actual systems

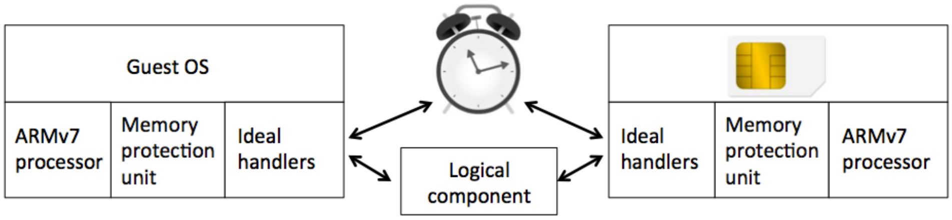

Figure A.1: Ideal model

behaviour. The key idea is to execute the partitions on physically separate, ideal processors, connected by an explicit, ideal communication channel, and equipped with a little extra paraphernalia, as shown in Fig. A.1. The ideal processors need to accurately mimick the execution of user space partitions on a real processor. This is done by augmenting the TLS processors by idealized functionality, the "ideal handlers" of Fig. A.1, which is invoked whenever the actual processors would transition to privileged mode by an exception (e.g. a hardware interrupt, or an exception). This construction allows userland code to execute as desired (with the exception of fine grained timing differences we currently do not take into account), but the idealized processors are physically prevented from directly affecting their sibling machine, with the exception of explicit communication using the message delivery service.

The task is thus to show that, properly set up, the user observable traces of the ideal model are the same as those of a "real model", obtained by executing the software in different partitions on top of our separation kernel, on top of a real ARMv7-A processor, including a Memory Management Unit (MMU) for physical protection of memory regions belonging to different partitions. We prove this using the bisimulation proof method [152], by exhibiting a concrete bisimulation relation, a *candidate relation*, relating the state spaces of ideal and real models. The proof that the candidate relation is actually a bisimulation relation of the appropriate type is in turn reduced to subsidiary properties, several of which have natural correspondences in previous kernel verification literature, cf. [90, 142], namely that:

i The initial states of the ideal and the real models are in the candidate relation. This is ensured by the correctness of the bootstrap code.

ii A partition does not perform any isolation-violating operation while it is executing. Due to our use of memory protection, this is really a noninterference-like property of the ARMv7-A architecture rather than a property of the separation kernel. This property is similar to the partition management result reported in [180].

iii The processor state switches correctly upon transition to privileged mode. Again this is a processor architecture rather than a kernel dependent result.

iv Execution of ideal exception handlers vs the real separation kernel exception handlers preserve the candidate relation.

v Several invariants are preserved while partitions and /or kernel handlers are running.

For the actual verification we have used a combination of theorem proving and binary code analysis. The real and ideal models are built on top of the Cambridge ARM HOL4 model, extended with a simple MMU unit. The isolation lemmas of ARM, items (ii) and (iii), are proved using a tool, ARM-prover, developed for the purpose in HOL4. The proofs are costly and involve traversing the full ARM instruction sets. The ARM-prover allows the proofs to be automated to a large extent. This frees us from the onerous task of verifying the two theorems on each element of the large ARM instruction set. The ARM-prover tool is developed on top of the monadic ARM semantics reported in [70]. Handlers (item (iv)) are verified using pre/post conditions. Manual generation of these handlers is an error-prone process, and for this reason we generate the pre/post conditions automatically based on the specification of the ideal model, the candidate relation, and the ARM isolation Lemmas. We transfer the pre/post conditions to the binary code analysis tool BAP [43], and use BAP to verify the bootstrap code and the kernel handlers at the assembly code level. Several tools have been developed for lifting the ARM code to the input format of BAP and manipulating the code. Space prevents us from more than outlining the ARM isolation lemmas and the BAP extensions here; these will be reported in separate publications.

The paper is organized as follows: In section 2 we present the ARMv7 processor, MMU, and timing model. In section 3, the PROSPER kernel is presented, and the the ideal model is described in section 4. We then proceed to give an overview of the proof strategy, including the decomposition of the TLS into the ARM lemmas, and the handler verification tasks. In section 6 we partially validate our verification approach by proving a "monotonicity of release" property as suggested by Sabelfeld and Sands [148], that a corollary of our proof is an NI property in the special case where partitions do not actually communicate. In section 7 we present the proof implementation, and in section 8 information is given on the status of the kernel and some performance figures regarding the proof itself. In section 9 related work is discussed, and finally in section 10, we conclude and discuss some unresolved issues.

A.2 ARMv7

An ARMv7 CPU has execution mode $m \in \mathcal{M}$ where

$$\mathcal{M} = \{usr, svc, abort, undef, irq, fiq, sys\}.$$

The non-privileged *usr* mode is used by the user partitions, while the privileged modes $\mathcal{M}_p = \mathcal{M} \setminus \{usr\}$ are used to execute kernel activities. A machine state is a record $\sigma = \langle regs, psrs, mem, coregs\rangle$ where *regs*, *psrs*, *mem* and *coregs*, respectively represent the registers, program status registers (*psrs*), memory and coprocessors of the machine. The register set *regs* consists of the sixteen user registers that are accessible in all modes as well as the banked-registers of each privileged mode that are available only in that mode. The program status registers is a record

$$\langle cpsr, psr_{svc}, psr_{abort}, psr_{undef}, psr_{irq}, psr_{fiq}, psr_{sys}\rangle$$

where *cpsr* is the current psr and each psr_m is the banked psr in mode $m \in \mathcal{M}_p$. A psr encodes the arithmetical flags, the executing mode, the interrupt mask, and the instruction decoding. The functions $I(\sigma)$ and $M(\sigma)$ return the hardware interrupt mask and the current mode in state σ, respectively. Moreover, the memory is the function $mem \in word32 \rightarrow word8$.

The tuple $coregs = \langle c_1, c_2, c_3\rangle$ contains the three 32-bit registers of coprocessor CP15, used mainly to control the Memory Management Unit (MMU). The register c_1 represents whether the MMU is enabled or not, and c_2 gives the base address of the page table. In ARMv7, there are sixteen domains, each representing a security role. The coprocessor register c_3 holds the current status of the domains. An entry of the page table determines the owner domain of the corresponding page and its access permission.

In our setting, a "real' system" is an ARM machine connected to a timer device. A real system is modeled by the record $s = \langle \sigma, t\rangle \in \mathcal{S}$, where σ is an ARM machine state and t represents the clock cycles elapsed since system start. The behavior of a system is defined by the state transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ where a transition is performed due to either the execution of an ARM instruction or a timer signal. We assume a simple time model that constrains all transitions to consume one clock cycle, i.e. if $\langle \sigma, t\rangle \rightarrow \langle \sigma', t'\rangle$ then $t' = t + 1$.

If the real system switches from the mode *usr* to a privileged mode, then an exception has occurred. The privileged mode *svc* is activated by a software interrupt (SWI). If the current instruction is undefined, then the system switches to the mode *undef*. The mode *irq* is activated by a hardware interrupt. In our setting, the timer triggers a hardware interrupt every fixed amount of clock (actually, instruction) cycles. If the MMU prevents an access to the memory, then the mode *abort* is enabled. In our setting, no exception can activate the modes *sys* and *fiq*. In fact, *sys* mode can only be explicitly entered from a privileged mode. Moreover, in our model there is only one device (the timer) which delivers standard hardware interrupts. For this reason the fast interrupt mode *fiq* is never activated. Whenever an exception occurs, the CPU backs up the program counter and the cpsr into the banked registers and into the psr of the activated mode, disables hardware interrupts and jumps to a predefined address in the vector table to transfer the control to the corresponding exception handler.

Figure A.2 (A) depicts an example computation of a real system. White and grey circles represent states in user mode and black circles represent states in priv-

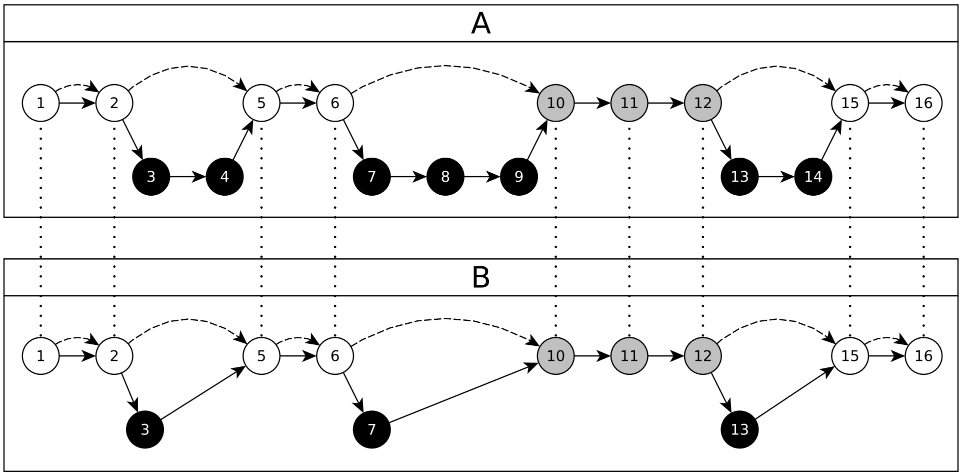

Figure A.2: (A) The real world and (B) the Top Level Specification

ileged modes. The circle labels represent the system clocks and the solid arrows represent the transition relation. Transitions between two states in user mode (e.g. $1 \rightarrow 2$) do not cause any exceptions. The timer tick of this example is six clock cycles, then an interrupt is delivered in the states 6 and 12, switching the system to mode *irq*. The transition between the states 2 and 3 is caused by a different exception, for example the execution of a software interrupt. Finally, transitions from privileged modes to user mode (e.g. $4 \rightarrow 5$) are caused by instructions that explicitly change *cpsr*.

A.3 The PROSPER Kernel

The PROSPER kernel has four minimal functionalities: execution of two user partitions on one physical machine, protection of the partition resources, partition scheduling, and inter-partition communication. All low-level tasks of the kernel that depend on the architecture (e.g. accessing special registers and coprocessors, context saving and restoring) are implemented in assembly ($\sim$ 150 lines of codes), while all high-level tasks (e.g., hypercall, scheduling, page table setup) are implemented in C ($\sim$ 600 lines of code). The current implementation can host OSs (e.g. μClinux [123], FreeRTOS[72]) that do not require intra-partition memory protection.

The machine memory is partitioned into three separate regions: the region in the range of $[min_g, max_g]$ for the partition $g \in \{1, 2\}$, and a kernel memory region. The accesses to the partitions are controlled by the MMU where three ARMv7 domains 0,1, and 2 are used to represent the kernel, the first partition and the second partition, respectively. When the kernel resumes a partition, it updates the coprocessor c_3 to set the partition domain to the value *client* and to disable the domain of the other partition. The MMU is configured to enforce the following properties: (i) if a partition is running, then only its memory can be accessed, (ii) whenever the kernel is activated (e.g. a partition performs a software interrupt), it is able to read and write its memory and the memory of the "interrupted" partition. We have no concurrency inside the kernel, i.e. an exception can not interrupt while another exception is being handled.

The partitions communicate through asynchronous message passing. Each partition has two executing status variables: *message status* is intended to process the incoming messages while the *task status* is used for other activities. To each status is associated a context that contains a set of user registers and the cpsr. For the active partition, the context corresponding to the active status is the current user registers and the cpsr and the context of the non-active status is stored in the kernel memory. For the inactive partition, both contexts are stored in kernel memory. The hypercalls are used by the partitions to invoke the kernel by executing the software interrupt instruction. The kernel provides two types of hypercalls: *message sending hypercall* and *status switching hypercall*. To send a message, the partition executes the instruction "SWI 1". The software interrupt handler stores the message into the message-box of the receiver and restores the sender. The status switching hypercall changes the executing status of the partition by executing "SWI 0". The kernel backs up the CPU state into its own memory and reactivates the interrupted partition.

The irq-handler implements a static round-robin scheduler, that suspends the active partition and resumes the other one. It is also in charge of delivering the pending messages to the resuming (receiver) partition; if the message-box of the resuming partition is full, (i) its status is changed to "message", (ii) the context of its message status is updated with the content of the pending message, and (iii) the program counter of the resumed partition is updated to point to its message handler code. The reception of a message causes the resumed partition to enter into a local critical section, i.e. no other message can be received while the partition is running in the message status. To exist from the critical section, the receiver partition performs a status switch hypercall.

To start the system, a memory image of the system must be prepared by the linker. Let $mem_1 : [min_1, max_1] \to word8$ and $mem_2 : [min_2, max_2] \to word8$ be two initial partition memories in our setting. Then, the linker loads the initial partition memories and the kernel memory into the system memory, and activates the kernel bootstrap code. The *initial state* of the real system is the first reachable state in user mode obtained after the execution of the bootstrap code of the kernel. Clearly, this initial state of the system depends on the partitions memories. We denote the behavior of a real system starting from an initial state s_0 with initial partition memories mem_1 and mem_2 by the transition system $T_r(mem_1, mem_2) = \langle \mathcal{S}, s_0, \to \rangle$.

A.4 The ideal system

The *ideal system* formalizes the top level specification which satisfies the required separation properties by construction. The ideal system is composed of two separate special ARMv7 machines communicating via asynchronous message passing, a logical component and a shared timer (see Fig. A.1). Each machine of the ideal system is used to execute one of the two partitions in a physically isolated environment. Intuitively, the logical component can be considered as an external device. Our special ARMv7 machine allows a partition to execute without the runtime support of the kernel. This machine executes the user-mode computations as a regular ARMv7 processor, but if the processor switches to a privileged mode, an abstract kernel functionality is atomically executed and the user mode is restored.

An ideal state is a record $q = \langle \sigma_1, \sigma_2, c_1, c_2, t, id \rangle \in \mathcal{Q}$ where t represents the clock cycles elapsed from the system start. At each instant, only one of the machines can perform computations, and $id \in \{1, 2\}$ identifies the active machine. The logical component consists of the records $c_i = \langle rdy, ctx, msg \rangle$, $i \in \{1, 2\}$. The flag rdy represents if the machine is ready to handle the incoming messages and the banked context ctx (set of registers and cpsr) is used to back up the machine state whenever a message is received. A message box $msg \in word32 \cup \{\bot\}$ can either contain a pending message or be empty ($\bot$). Henceforth, we say that an ideal system is in mode m if the active machine is in mode m and the inactive one is in the mode usr.

The initial state of the ideal system, similar to the real system, depends on the initial partition memories. We denote the behavior of a ideal system starting from an initial state q_0 with initial partition memories mem_1 and mem_2 by the transition system $T_i(mem_1, mem_2) = \langle \mathcal{Q}, q_0, \rightarrow \rangle$. In the initial state, all the components are initialized such that the partitions memories and the page tables are loaded into their corresponding machine memory, the program counter of each machine points to the entry point of the corresponding partition, and both machines are in the mode usr.

Figure A.3 shows the semantics of the ideal system when machine 1 is active. Due to lack of space, we only present the semantics of hypercalls and scheduling. In all cases, the ideal transitions yield the two machines in user mode. The rules for machine 2 active are similar. Each kernel functionality f comes with a fixed time budget t_f for its execution.

The rule *UserR* states that if the processor is in user mode, execution of an instruction does not affect the state of the logical component and the inactive machine, and it behaves as if it is executed on a regular ARMv7 machine. Instructions consume the same amount of cycles in the real and ideal systems.

A machine is rescheduled whenever a hardware interrupt is triggered by the timer. The function *RestoreUser* restores user mode and the corresponding banked registers. The rule *SchR* describes that if there is no message for the resumed machine or the resumed machine is not ready to handle a message, the active partition is changed to the current suspended one. The rule *RcvR* expresses the

$$\frac{M(\sigma_1) = \mathit{usr} \wedge \langle \sigma_1, t\rangle \to \langle \sigma_1', t'\rangle}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \to \langle \sigma_1', \sigma_2, c_1, c_2, t', 1\rangle}\ \mathit{UserR}$$

$$\frac{M(\sigma_1) = \mathit{irq} \wedge (c_2.\mathit{msg} = \perp \vee \neg c_2.\mathit{rdy})}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \to \langle \mathit{RestoreUser}(\sigma_1), \sigma_2, c_1, c_2, t + t_{sh}, 2\rangle}\ \mathit{SchR}$$

$$\frac{\begin{array}{c} M(\sigma_1) = \mathit{svc} \wedge \mathit{curr}(\sigma_1) = \mathit{SWI\ 0} \\ (\sigma_1', c_1') = \mathit{Switch}(\mathit{RestoreUsr}(\sigma_1), c_1)\end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \to \langle \sigma_1', \sigma_2, c_1', c_2, t + t_{switch}, 1\rangle}\ \mathit{SwitchR}$$

$$\frac{\begin{array}{c} M(\sigma_1) = \mathit{irq} \wedge c_2.\mathit{rdy} \wedge c_2.\mathit{msg} \neq \perp \\ (\sigma_2', c_2') = \mathit{Receive}(\sigma_2, c_2)\end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \to \langle \mathit{RestoreUsr}(\sigma_1), \sigma_2', c_1, c_2', t + t_{rcv}, 2\rangle}\ \mathit{RcvR}$$

$$\frac{\begin{array}{c} M(\sigma_1) = \mathit{svc} \wedge \mathit{curr}(\sigma_1) = \mathit{SWI\ 1} \\ c_2' = \langle c_2.\mathit{rdy}, c_2.\mathit{ctx}, \mathit{Out}(\sigma_1)\rangle\end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \to \langle \mathit{IncPC}(\mathit{RestoreUsr}(\sigma_1)), \sigma_2, c_1, c_2', t + t_{snd}, 1\rangle}\ \mathit{SendR}$$

Figure A.3: Semantics of the Ideal System

reception of a pending message by the resumed machine in which the function *receive*(σ_2, c_2) disables the *rdy* flag for entering into a critical section, moves the pending message into the input buffer of the receiver, and sets the program counter to point to the message handler of the receiver.

The rule *SendR* describes the semantics of message sending (activated due to the execution of "SWI 1" in the previous state) that copies the message into the message box of the inactive machine. If the status switch hypercall is invoked, the rule *SwitchR* toggles the *rdy* flag and restores the banked context.

Figure A.2 (B) depicts an example computation of an ideal system. In the states 2, 6 and 12 the system traps an exception raised on the active machine and atomically applies an ideal kernel functionality.

A.5 Proof Strategy

To prove that the real model does not introduce information channels not already present in the ideal model it suffices to show that the observable traces for each partition are the same in both cases. In order to pin down this concept we need to define when each partition system is in control of the system, and what its observations are.

Top Level Proof Goal Intuitively, for the real system, partition g is in control when its program counter points to a location in mem_g. However, this is not entirely accurate. Instead we say that partition g is *active* in state s, $act_g(s)$, if the processor is in user mode and the status of the domain g, held by coprocessor register c_3 is client. For the ideal system, partition g is *active*, $act_g(q)$, if the *id* field of the ideal state is g and the corresponding machine is in the user mode.

The observations of partition g in real state s, assuming that g is active, are the CPU and memory resources observable by g in state s. This is the structure $O_g(s) = \langle uregs, cpsr, mem_g \rangle$ of user registers, cpsr and partition memory in state s. If g is inactive in s, g's observations are the user registers and cpsr of the saved context of g along with its partition memory. For the ideal system, $O_g(q)$ is the user registers, cpsr and the memory allocated to g of the corresponding machine in q.

Consider now an infinite execution $\pi_r = s_0 \rightarrow s_1 \rightarrow \cdots$ of the real system. The g-trace of π_r is the sequence $\omega(\pi_r, g)$ of observations obtained by first projecting out those states for which g is not active, and secondly extracting g's observations, or in other words, $\omega(\pi_r, g) = map(O_g, prj(\pi_r, act_g))$ where *prj* and *map* are the obvious projection/filtering functions. Similarly, if π_i is an ideal execution, the g-trace of π_i is $\omega(\pi_i, g) = map(O_g, prj(\pi_i, act_g))$.

Let now $tr_{g,r}(mem_1, mem_2)$ and $tr_{g,i}(mem_1, mem_2)$ be the set of g-traces of the real system and the ideal system with the initial partition memories of mem_1 and mem_2, respectively, and for arbitrary states s, q, let $tr_g(s)$ and $tr_g(q)$ be the sets of g-traces of executions starting in s, q, respectively. The top level proof goal is thus to prove that the sets of g-traces of the real and the ideal systems are identical, for $g \in \{1, 2\}$ and any arbitrary mem_1 and mem_2, or, more precisely, that

$$tr_{g,r}(mem_1, mem_2) = tr_{g,i}(mem_1, mem_2) \tag{A.1}$$

for all initial partition memories mem_1, mem_2. If (A.1) holds we say that the PROSPER kernel guarantees isolation.

In order to prove (A.1), we first present three general lemmas concerning the safe executions of an ARM machine in user mode and its safe mode switching from user mode to privileged mode. Given the general ARM Lemmas, we prove lemmas for the real and the ideal systems to ensure correct initialization, correct userland execution, and isolation-guaranteed execution of the kernel handlers. We then proceed to present the proof of (A.1).

ARM Lemmas Our proof strategy identifies three lemmas concerning the ARM instruction set architecture that may have significance beyond the verification exercise reported here. The first is a general noninterference lemma stating that if an ARM machine executes in user mode in a memory protected configuration as studied here, the behavior of the active partition is influenced only by those resources allowed to do so. The predicate $sim_g(s_1, s_2)$ indicates that the status of the domain g held by coprocessor register c_3 is client in the user mode states s_1 and s_2, and they have the same user registers, cpsr, MMU configurations and the memory allocated to the domain g.

Lemma 1. If $sim_g(s_1, s_2)$ and $s_1 \to s_1'$, there exists s_2' s.t. $s_2 \to s_2'$ and $sim_g(s_1', s_2')$, and vice versa.

The second lemma establishes $\mathit{unmodified}_g(s, s')$ stating that the non-accessible resources for a state s in user mode, including the privileged psrs/registers, coprocessor registers, interrupt flags and the memory regions not allocated to the active partition g, are not modified in a transition from s to another user mode state s'. If s' is in privileged mode m, the privileged registers, psr_m and the interrupt flags, are excluded from the non-modifiable resources. We obtain:

Lemma 2. If $s \to s'$ and $act_g(s)$ then $\mathit{unmodified}_g(s, s')$.

The predicate $priv\_const_g(s, s')$ asserts that if an ARM machine switches from s in user mode to s' in privileged mode m then the conditions for the execution of the handler are prepared properly, e.g., the program counter points to the correct entry of the vector table, the link register of m contains the correct return address of the partition, and the flags of $cpsr$ and psr_m are set correctly.

Lemma 3. Suppose $s \to s'$, $act_g(s)$ and $M(s') \neq usr$. Then $priv\_const_g(s, s')$.

(g, m)-Compatible States We then turn to the conditions needed to ensure that the partition observations are the same in the real and the ideal systems. These conditions depend on many aspects of the machine states and we are only able to outline the conditions here.

These conditions are complex because several elements can directly or indirectly influence the behavior of a partition. We briefly define the conditions for the user mode states (i.e. when all machines are in the user mode) and the switched mode states (i.e. the inactive machine of ideal machine is in the user mode but the real system and the active machine of ideal system have recently switched to the privileged mode).

Say that two states s and q are (g, m)-*compatible* if (i) s and q are in the mode m, (ii) g is the last active partition in s and q, i.e. the status of the domain g, held by coprocessor register c_3 is client in s, and the *id* field of the ideal state is g, (iii) the partitions have the same observations in s and q, $O_{g'}(s) = O_{g'}(q)$ for $g' \in \{1, 2\}$, (iv) the values of data-structures in the logical component of state q agree with the values of corresponding data structures in the kernel of state s, e.g. the message box of a partition in the kernel and the logical component contain the same values, (v) the MMU and coprocessors are configured correctly in all three machines, (vi) a set of invariants are held by the kernel data-structure in s, (vii) the interrupt flags are set correctly, e.g. the fast interrupt flag F of all machines are disabled, (viii) if m is a privileged mode then psr_m and the link register of the mode m must be identical in the active machine of q and s, to make sure that g is restored properly, (ix) s and q have the same system clock.

User/Handler Lemmas The relation of (g,m)-compatibility is our candidate unwinding relation, i.e. it is in some suitable sense which we go on to make precise preserved under computation. This involves showing the following two key properties:

- User Lemma: Each user mode transition in the real system is matched (in the sense of (g,m)-compatibility) by a corresponding user mode transition in the ideal system without interfering with the resources that are not intended to be accessible by the partition, and vice versa.
- Handler Lemma: Each complete handler execution in the real system is matched (as (g,m)-compatibility) by a corresponding execution of a kernel functionality in the ideal model, and vice versa.

Similar to the $\mathit{unmodified}_g$ predicate for the real system above, $\mathit{unmodified}_g(q, q')$ holds if the logical component, the inactive machine, and the non-accessible resources of the active machine are unmodified by an ideal transition from q to q' when g is active in q. The User Lemma then follows from the first and second ARM Lemmas as follows:

Lemma 4 (User)**.** For all (g, usr)*-compatible* states s and q, if $q \rightarrow q'$ then there exist s' and m such that

(i) $s \rightarrow s'$,

(ii) the states s' and q' are (g, m)*-compatible*,

(iii) $\mathit{unmodified}_g(s, s')$, and (iv) $\mathit{unmodified}_g(q, q')$.

Vice versa, if $s \rightarrow s'$ then q' and m exists such that $q \rightarrow q'$ and the above properties (ii) and (iii) hold.

For the Handler Lemma we need to ensure that execution of the kernel handlers terminates, and that the compatibility conditions are satisfied when the control returns back to the partitions. Let $s_0 \rightsquigarrow s_n$ if there is a finite execution $s_0 \rightarrow \cdots \rightarrow s_n$ such that $M(s_n) = usr$ and $M(s_j) \neq usr$ for $0 < j < n$. Similarly, we define $\rightsquigarrow$ for the ideal system. These state relations are represented in Fig. A.2 by the dashed arrows. The additional black states in the real world are internal kernel steps that can not be observed by the partitions.

Lemma 5 (Handler)**.** Suppose that s and q are two (g, m)*-compatible* states, $m \neq usr$. Assume s and q are respectively reached by a transition from the states s' and q', and $priv\_const_g(q, q')$ and $priv\_const_g(s, s')$ hold. If $q' \rightsquigarrow q''$ then there exist s'' and $g' \in \{1, 2\}$ such that $s' \rightsquigarrow s''$, and the states s'' and q'' are (g', usr)*-compatible*, and vice versa.

Furthermore, it is to be guaranteed that the initial states of the real and the ideal systems are compatible. That is, we must verify that the MMU is set up according to our model requirements, the kernel invariants are satisfied, the partitions memory and the interrupt flags are configured correctly. In addition, we must make sure that the kernel code is loaded in the right part of the memory.

Proposition 1. *For all initial partition memories mem_1, mem_2, the kernel boot terminates in the state s_0 and there exists $g \in \{1, 2\}$, such that s_0 and q_0 are (g, usr)-compatible.*

Proof of Main Theorem We can now proceed to prove (A.1). This is almost done once we show that the initial states are related by a bisimulation relation of a suitable form. To this end say that a relation R on pairs (s, q) of user mode states is a *candidate relation*, if whenever sRq then for some $g \in \{1, 2\}$, (i) $act_g(s)$, (ii) $act_g(q)$, (iii) $O_g(s) = O_g(q)$, and (iv) if $q \rightsquigarrow q'$, then there exists s' such that $s \rightsquigarrow s'$ and $s'Rq'$, and (v) vice versa, if $s \rightsquigarrow s'$, then there exists q' such that $q \rightsquigarrow q'$ and $s'Rq'$.

Note that for the user mode states s and s', if $s \rightarrow s'$ then $s \rightsquigarrow s'$. In Fig. A.2, dotted lines exemplify a bisimulation relation. It is easy to check that the existence of a candidate relation is sufficient to ensure (A.1). In particular:

Proposition 2. *Suppose that sRq for some candidate relation R. Then $tr_g(s) = tr_g(q)$.*

Theorem 2. *The PROSPER separation kernel guarantees isolation.*

The proof of 2 obviously relies on the proofs of the above lemmas, which in turn, for Handler Lemma and the Lemma 1, relies on verification of the handler and the bootstrap code, as outlined in Section A.7. But, it may be illustrative to explain how these lemma come together to allow the main Theorem 2 to be proved.

By Theorem 2 it suffices to find a candidate relation relating the initial states s_0 and q_0. Define the candidate relation as follows:

$$R = \{(s, q) | (g, usr)\text{-}compatible(s, q) \wedge g \in \{1, 2\}\}$$

We get s_0Rq_0 by prop. 1.

To prove that R is a candidate relation, assume that sRq. Then s and q are (g, usr)*-compatible*. Thus, $act_g(s)$, $act_g(q)$ and $O_g(s) = O_g(q)$ hold.

Suppose now that $q \rightsquigarrow q'$. There are two cases:

Case 1: If the ideal transition does not involve mode switching it follows from the User Lemma that s' exists such that $s \rightsquigarrow s'$ and q' and s' are (g, usr)-*compatible*, whence $s'Rq'$ as desired.

Case 2: If the ideal transition involves a switch to privileged mode m, it follows from the User Lemma that the real and ideal system evolve to the (g, m)-*compatible* states s'' and q''. According to the Third ARM Lemma, these transitions are performed safely, i.e. $priv\_const_g(s, s'')$ and $priv\_const_g(q, q'')$ hold. From the Handler Lemma, we can conclude that there exist (g', usr)-*compatible* states s' and q' where $s'' \rightsquigarrow s'$, $q'' \rightsquigarrow q'$ and $g' \in \{1, 2\}$. But then $s'Rq'$, as desired.

The converse direction, that $s \rightsquigarrow s'$ implies $q \rightsquigarrow q'$ and $s'Rq'$ for some q follows by a symmetric argument (or, in this simple case, by determinacy of the $\rightsquigarrow$ relation). This concludes the proof of Theorem 2.

A.6 Isolation Properties

The main theorem shows that the real system does not leak more information than the ideal system, under the caveats we have imposed. However, it may not be clear what information is leaked by the ideal system itself. Neither may it be clear how leakage properties of the ideal system can be transferred to the real system. In this section we throw light on these two issues.

Data Separation Concerning the ideal system itself we use the approach of [90] to analyze kernel data separation properties. Let $\mathcal{Q}_c = \{q | \exists s.\ sRq\}$ be the image of the candidate relation.

The *No-Exfiltration* property guarantees that a transition with the partition g active in its target, does not modify the resources of the other partition, except its communication channel, i.e. the message box:

Lemma 6. Let $g, g' \in \{1, 2\}$, $g' \neq g$ and $q \in \mathcal{Q}_c$. If $q \rightsquigarrow q'$ and $q'.id = g$, then $O_{g'}(q) = O_{g'}(q')$, $q.c_{g'}.rdy = q'.c_{g'}.rdy$ and $q.c_{g'}.ctx = q'.c_{g'}.ctx$.

The *No-Infiltration* property is a noninterference property guaranteeing that a transition for which g is active in its target state, depends only on the partition observations, its logical component and the MMU configuration. In particular, a transition ending in a state with the partition g active, is not influenced by data owned by the other partition.

Lemma 7. Let $q_1, q_2 \in \mathcal{Q}_c$ such that $q_1.c_g = q_2.c_g$, $q_1.t = q_2.t$, $q_1.id = q_2.id$ and $O_g(q_1) = O_g(q_2)$. If $q_1 \rightsquigarrow q_1'$, $q_2 \rightsquigarrow q_2'$, $act_g(q_1')$ and $act_g(q_2')$ then $q_1'.c_g = q_2'.c_g$, $q_1'.t = q_2'.t$ and $O_g(q_1') = O_g(q_2')$

Similar properties can be proven for the real system using the candidate relation and the properties of the ideal system. Let $\mathcal{S}_c = \{s | \exists q.\ sRq\}$ be the preimage of the candidate relation, and the function $lc_g(s)$ extracts from the kernel memory the content of the data-structure that corresponds to c_g in the logical component. The following corollary states the no-infiltration and no-exfiltration for the real system.

Corollary 1. Let $s_1, s_2 \in \mathcal{S}_c$, $s_1.t = s_2.t$, $act_g(s_1) \Leftrightarrow act_g(s_2)$.

- Suppose if $g' \neq g$, $s_1 \rightsquigarrow s'_1$ and $act_g(s'_1)$ then $O_{g'}(s_1) = O_{g'}(s'_1)$, $lc_{g'}(s_1).rdy = lc_{g'}(s'_1).rdy$ and $lc_{g'}(s_1).ctx = lc_{g'}(s'_1).ctx$.
- Suppose if $lc_g(s_1) = lc_g(s_2)$, $O_g(s_1) = O_g(s_2)$, $s_1 \rightsquigarrow s'_1$ and $s_2 \rightsquigarrow s'_2$, $act_g(s'_1)$ and $act_g(s'_2)$ then $lc_g(s'_1) = lc_g(s'_2)$ and $O_g(s'_1) = O_g(s'_2)$

We sketch the proof of the second statement. Since s_1 and s_2 are in $\mathcal{S}_c$, then there exist q_1 and q_2 s.t. $s_1 R q_1$ and $s_2 R q_2$. We follow from the assumptions and the definition of R that $q_1.c_g = q_2.c_g$, $q_1.t = q_2.t$, $O_g(q_1) = O_g(q_2)$ and $q_1.idx = q_2.idx$. Since the candidate relation is a bisimulation, then for $j = 1, 2$, there exists q'_j s.t. $q_j \rightsquigarrow q'_j$ and $s'_j R q'_j$. Thus, $act_g(q'_j)$, $lc_g(s'_j) = q'_j.c_g$ and $O_g(s'_j) = O_g(q'_j)$. We conclude the proof by showing that $O_g(q'_1) = O_g(q'_2)$ and $q'_1.c_g = q'_2.c_g$ according to the Lemma 7.

Noninterference The no-exfiltration/no-infiltration properties give limited data separation properties at the level of single transitions. They do not, however, lift to executions, because messages may be passed between partitions which can introduce explicit data dependencies. As a sanity check, we therefore show how the security condition is reduced to noninterference in the special case where no exception except timer signal takes place. This property formalizes the intuition that if the partition g does not communicate, then the execution of the other partition is completely independent of activities of g.

A partition with the initial memory mem is called non-communicating, if for all arbitrary mem' and all states q that are reachable from the initial state of $T_i(mem, mem')$, $M(q.\sigma_1) = \{usr, irq\}$ holds.

Theorem 3. For any two non-communicating partitions with the initial memories mem_1 and mem'_1, and an arbitrary partition with the initial memory mem_2,

$$tr_{1,i}(mem_1, mem_2) = tr_{1,i}(mem'_1, mem_2)$$

The symmetric theorem is proved when the non- communicating partition is deployed on the second machine. The state of a machine can be externally changed only by the reception of a message. Since the non-communicating partition never raises an exception, it can not execute the software interrupt and it can not send a message. Moreover, the system clock, shared between the two machines, must be independent of the activity performed by the non-communicating machine. This is possible because we assume that all transitions, with the exception of the ideal functionalities, require one clock cycle. Note that the candidate relation allows Theorem 3 to be directly transferred to the real system. The details are left out.

A.7 Proof Implementation

The overall proof is carried out in the HOL4 theorem prover, following the proof strategy presented in Section A.5. For those parts (the Handler Lemma and Proposition 1) that depend on kernel code we generate contracts and transfer the verification to BAP. To realize this we have produced a number of helper tools of which the main ones are: (i) an ARMv7 prover tool implemented in SML/HOL4, (ii) various tools and tool components interfacing HOL4 with BAP, (iii) a lifter tool to convert ARM assembly to BAP's input language. Space prevents us from more than outlining the BAP extensions and the proof of the ARM Lemmas here; these will be reported in separate publications.

A.7.1 Verification in HOL4

Overview of the ARM Model We use Fox et al's monadic HOL4 model of the ARMv7 instruction set architecture. The model has been validated against a development board, giving some credence to its accuracy [70].

A *computation* in the monadic HOL4 ARM model is a term of type

$$\alpha\ M = \texttt{arm\_state} \mapsto (\alpha \times \texttt{arm\_state})\texttt{error\_option}.$$

Computations act on a state `arm_state` and return either `ValueState` $a\ s$, a new state s of type `arm_state` along with a return value a of type α, or an error `Error` e. Errors represent all unpredictable computations, i.e., those that are underspecified by the ARM specification. The monad unit injects a value into a computation, while binding is a sequential composition operation which passes the return value of the first computation to the input parameters of the second one as follows:

$$\begin{array}{ll} \texttt{f} \gg= \texttt{g} = (\lambda\texttt{s.} & \texttt{case f s of Error e} \mapsto \texttt{Error e} \\ & \texttt{|| ValueState y t} \mapsto \texttt{g y t)} \end{array}$$

The execution of an ARM instruction is defined by the computation `arm_next` modeling the entire processing of an instruction, from fetching the instruction pointed to by the program counter to the actual instruction execution.

The MMU Extension We extend the ARM model to support the MMU functionality in our setting. Given the complexity of memory management, the model is restricted to support only those parts of the MMU functionality used by the PROSPER kernel.[2] We also proved that the MMU configurations of all reachable states are supported by the extended model and not underspecified. The original ARM model tracks the history of memory accesses, allowing to compute the set of memory pages accessed by an instruction. To be accurate, it is necessary to check

[2]Only section-based one-level page tables without address translation are supported so far.

the access list after each primitive computation. To this end, the monadic structure is modified so that access history checks are introduced at every sequential composition of two computations. In case of an access violation within the first computation, the second one is simply disregarded, returning the unspecified value *ARB* along with the first state where an access violation has been recorded.

```
f ≫= g = (λs.  case f s of Error e ↦ Error e
               || ValueState y t ↦
               (if (access_violation t)
               then (ValueState ARB t)
               else (g y t)) )
```

Proof of the ARM Lemmas We use a relational Hoare logic framework to prove the ARM Lemmas. For technical reasons we formulate the three ARM Lemmas as a single statement. For any computation `f` and predicates $\mathtt{p}_1$, $\mathtt{p}_2$, we define a relational predicate $\mathtt{preserving}(\mathtt{f}, \mathtt{p}_1, \mathtt{p}_2)$ stating that, when starting from two states in the relation sim_g and satisfying $\mathtt{p}_1$, then the states returned by `f` (i) are in the relation sim_g, (ii) satisfy the non-modification and mode-switching constraints, as presented in Section A.5, and (iii) satisfy $\mathtt{p}_2$. The state predicates $\mathtt{p}_1$ and $\mathtt{p}_2$ allow processor mode specific reasoning. The final goal is to show that the MMU-enabled variant of `arm_next` satisfies `preserving` when starting from user mode.

A set of sound inference rules have been implemented in a semi-automatic HOL4 helper tool. An example is the rule for sequential composition:

$$\frac{\mathtt{preserving}(\mathtt{f}_1, \mathtt{p}_1, \mathtt{p}_1) \quad \mathtt{preserving}(\mathtt{f}_2, \mathtt{p}_1, \mathtt{p}_2)}{\mathtt{preserving}(\mathtt{f}_1 \gg= (\lambda \mathtt{x}.\mathtt{f}_2), \mathtt{p}_1, (\mathtt{p}_1 \vee \mathtt{p}_2))}$$

The tool recognizes the structure of a computation, decomposes the verification goal in a set of sub-goals, proves the sub-goals recursively and applies the suitable inference rule to infer the initial goal. Moreover, it searches in the HOL4 database and the user-provided theorems to find a suitable theorem that can prove the goal. We prove `preserving` for the write primitive computations manually, but the tool can handle some read computations automatically, allowing to prove a large share of the workload automatically.

Generation of Pre- and Postconditions HOL4 is also used to generate pre- and postconditions for the kernel handlers, for subsequent verification with BAP. Consider a handler with the starting state s_1 in mode m such that $s_1 \rightsquigarrow s_2$. Suppose that s_1 and q_1 are (g, m)-compatible such that q_1 is the starting state of the corresponding ideal handler functionality. Let q_2 be a state such that $q_1 \rightsquigarrow q_2$. These conditions allow to automatically generate the precondition of the handler under which the final state s_2 will be (g, usr)-compatible with q_2. The preconditions are generated by the hypotheses of the Handler Lemma: the starting state of the kernel handler s_1 is (g, m)-*compatible* with q_1, and there are s_0 and q_0, such that $priv\_const_g(q_0, q_1)$, $priv\_const_g(s_0, s_1)$, and s_0 and q_0 are in the candidate relation.

A.7.2 Binary Code Verification

The kernel code verification relies on Hoare logic. To prove the Handler Lemma and Proposition 1, we are required to verify several Hoare triples $\{P\}C\{Q\}$ for the exception handlers and the bootstrap code, that is we check that if the precondition P holds in the starting state of C, then the postcondition Q is guaranteed by C. When possible, we adopt a standard semi-automatic strategy, i.e. firstly, we compute the weakest liberal precondition $wlp(C, Q)$ on the starting state, then prove that the precondition P implies the weakest precondition. This task can be fully automated if the predicate $P \implies wlp(C, Q)$ is equivalent to a predicate of the form $\forall x.A$ 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 [74].

Weakest preconditions can be computed directly in HOL4 using the ARMv7 model. However, this task requires a significant engineering effort. We adopted a more practical approach, by using (BAP) [43]. The BAP toolset provides platform-independent utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to perform wp 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.

We found the existing BAP front-end to translate ARM programs to BIL inadequate for our purpose: It supports only ARMv4, it does not manage the processor status registers, and it does not handle banked registers for the privileged modes and coprocessor management. To this end, we developed a new front-end for ARMv7 programs using the ARM model available in HOL4. This tool allows us to translate the code of the kernel handlers and the bootstrap into BIL.

The HOL4 ARM model provides the function `arm_steps` to compute the set of pairs $\langle c_1, t_1\rangle, \ldots \langle c_n, t_n\rangle$ for an instruction where the function t_i transforms a state provided that the condition c_i holds on that state. In other words, $\forall$`s:arm_state arm_next s` = `ValueState ()` t_i(`s`) if c_i(`s`). In order to use `arm_steps`, the execution mode and the instruction set type (e.g. Thumb, ARM) must be known. Our handlers preconditions set the value of these parameters. The translation from ARM to BIL is performed by translating the HOL4 conditions c_i and functions t_i to BIL fragments.

Verifying the Hoare triples using weakest preconditions requires us to handle some common issues. Algorithms to compute weakest preconditions rely on the absence of indirect jumps, i.e. the jumps whose target address is a variable. We used the SMT solver to automatically compute jump targets, depending on the instruction precondition. Weakest preconditions can grow exponentially in the number of instructions. We extended BAP to simplify the weakest precondition during its backward propagation using ARM specific simplification patterns. Finally, our verification task has been simplified by the structure of the kernel code. All loops of the kernel have a single control flow node that represents both the entry point and the exit point of the loop. In the general case, we defined a loop invariant and a loop variant and applied the standard Hoare logic rules to prove the contract. Verification has been simplified by the absence of loops in the kernel handlers and the fact that the boot code contains only for-loops that iterate over integer sets.

A.8 Evaluation

We tested the kernel implementation using OVP [133] as main execution environment, which provides a simulation infrastructure convenient to evaluate our kernel. Slightly different versions of the kernel have been deployed on Beagleboard, Beagleboard-XM, Beaglebone, NovaThor and the Integrator development board. The size of kernel code, internal data-structures and page table are respectively less than 4 kB, 2 kB and 16 kB. The main functionality of the kernel is provided by the software and hardware interrupt handlers. In the worst case, the hardware interrupt handler executes 112 instructions, including 48 reading and 22 writing accesses to the memory. Similarly, the software interrupt handler executes at most 46 instructions, including 20 reading and 8 writing accesses to the memory. All the memory locations accessed by the kernel handlers belong to the internal kernel data-structures. To minimize the system overhead and avoid accesses to system memory during kernel tasks, we can use scratchpad memory or cache locking due to the size of the run-time footprint.

We identified and fixed several bugs in the kernel implementation during the verification process: (i) the registers were not sanitized after the bootstrap, (ii) some of the execution flags were not correctly restored during the context switch, (iii) the procedure to decode the hypercall identifier did not consider the case that the partition is running in thumb execution mode.

The model of the ideal system, the formalization of the verification procedure and the proofs of the theorems consist of 21k lines of HOL4 code. The tools developed to support the verification of the kernel contracts required 2k lines of HOL4 and python code. The kernel binary code is verified with respect to sixteen contracts, each of them consisting of ~ 400 lines of assertions that are automatically generated from HOL4. In the worst case, the verification of one contract required $\sim$ 30 minutes using one Intel(R) Xeon(R) X3470 core; the contract is generated in ~ 5 minutes, the indirect jumps are solved in ~ 2 minutes, the weakest precondition is computed in ~ 10 minutes and the SMT solver verifies the validity of the resulting condition in ~ 15 minutes.

A.9 Related Works

Past work on formal verification of kernel information flow properties [90, 124, 142, 25] are based on variants of noninterference [77]. Typically, the goal is to allow a number of component systems, partitions, or guest systems, depending on terminology, to share a computing platform without any interaction, leaving possible communication between the partitions to be managed by mechanisms outside the model. In Heitmeyer et al [90], for instance, partitions have explicit input and output buffers, but communication is delegated to external agents, in this way allowing properties like absence of infiltration (roughly: direct flows) and exfiltration (indirect flows) to be proved. Similar results are reported in [142, 25] and in [180] at the firmware level. Murray et al [125] considers noninterference in presence of a dynamic scheduler and uses a version of intransitive noninterference [145] (actually, NI) to allow a scheduler to influence which partition is scheduled, without permitting the scheduler to be used as a covert channel, as discussed briefly in the introduction.

Several recent works address hypervisor/microkernel verification, although without taking information flow into account. In [105] a simulation property of an entire microkernel down to a C implementation was verified using the Isabelle theorem prover. This work was recently extended to ARM assembly using decompilation techniques [156]. Alkassar et al [5] proposed an automatic approach to verify a hypervisor for a (simplified) MIPS machine by annotating the C code with contracts and checking them using VCC. They establish a reachability property: at any time the state of a partition can be reached by executing the same partition on a completely isolated machine. This is sufficient to establish simulation when the specification is deterministic (but not otherwise). To allow VCC to reason about statically unknown partitions/guests, a C emulator of the MIPS machine has been implemented and annotated. The C emulator has been adopted also to verify parts of the hypervisor that mix C and assembly code [134].

Most kernel security analyses address the kernel routines one at a time, using suitable relational specifications. Without verifying the correct interaction between the kernel routines and the processor (e.g. mode switching and memory protection), these specifications are not sufficient to guarantee security at system level, i.e. at the level of “full” executions that interleave kernel routines with userland execution of the partitions. Performing such a systems-level, integrated analysis (kernel and processor) has not been done before for realistic processor architectures. For instance [180, 124] address kernel routines but not the processor interaction. Analysing each kernel routine in isolation can be done using existing versions of conditional non-interference (as discussed in [124]). However, this does not guarantee information flow security at system level. Our approach to formulating the TLS using idealized userland processors solves this problem, simply by showing that the full executions for the ideal and the real model are the same.

Barthe et al. [25] formalized a hypervisor model using the Coq proof assistant. They focus on establishing that the hypervisor ensures isolation properties between the guests, abstracting away from actual hypervisor implementation.

A.10 Discussion

We have presented a separation kernel, the PROSPER kernel, for ARMv7-A and a machine-assisted proof of information flow correctness using a combination of tools (HOL4 [95] and BAP [43]). Our analysis has a number of distinguishing features:

- Our top-level specification (TLS) and verification approach is deliberately designed to take inter-component communication into account, an ever present challenge in the verification of information flow properties for real systems.

- We introduce a new technique for building a TLS for this type of application, based on communicating idealized userland processors.

- The security analysis is performed at systems level, modeling both the MMU-constrained user space execution of arbitrary partitions, the kernel handlers, and the interaction of the two.

- We validate the "monotonicity or release" property, as suggested by Sabelfeld and Sands [148], by showing that the security proof reduces to standard non-interference for the special case of non-communicating partitions.

- The entire analysis is performed at machine code level for a commodity processor architecture.

A number of subsidiary contributions include several tools for managing and executing the proofs, including the ARM prover tool for verifying critical partition correctness properties of the ARMv7 machine architecture based on an extension of Fox and Myreen's monadic ARM semantics [70], and several extensions to the BAP toolset.

By verifying the entire kernel at machine code level we avoid reliance on a C compiler, and we can transparently verify code that mix C and assembly. Generally speaking, verification at machine code level is time consuming, however we were supported by the fact that the code was mostly compiler produced and loops were used in only a few places.

Since our TLS specifies the exact set of traces allowed by an implementation, a worry might be that the TLS becomes overly detailed and unwieldy. We did not find this to be the case. Rather, the development of the ideal model, as we progressed to understand the various issues involved, was a great help in organizing the thinking. It is true that our approach (as in other work, cf. [90]) precludes an abstract treatment of scheduling, but this is to be expected when information flow is to be taken into account.

On two counts our model is not yet satisfactory. The first concerns timing. Our model counts instruction cycles [3], instead of real clock cycles. In our implementation the former is used. It is non-trivial to extend our analysis to a more realistic time representation, as in this case well-known phenomena such as cache delays and instruction pipelining come into play. Cache leakage has been considered in the context of virtualization by Barthe et al [26]. Zhang et al [185] demonstrated an access-driven side-channel attack targeting the Xen hypervisor. The authors (i) use interprocess interrupts to affect the Xen scheduler and to reduce the time slot available to the victim and (ii) indirectly monitor the usage of the instruction cache, which is shared among partitions. Extending our approach to handle access-driven attacks requires a more refined analysis of timing behaviour, which is part of our ongoing research efforts.

The second count is unpredictable states. According to the ARM Architecture Reference Manual [13], unpredictable behaviour is not allowed to "perform any function that cannot be performed at the current or lower level of privilege using instructions that are not unpredictable". This definition is difficult to accommodate in our framework. An interpretation of allowed behaviour which is adequate for our purpose is "compliant with the ARM Lemmas". This enables our proofs to go through, and in fact we posit that this may be a more helpful and less prescriptive definition than that of [13]. Practically, the ARM Lemmas can be used to certify if a specific ARMv7-A implementation can be used to host our kernel. In the proof implementation we have used the error states introduced in the monadic ARM HOL4 model. This is not really satisfactory, though, as this allows partitions to exit the scope of our model at will, by entering an unpredictable state. We leave a better treatment of unpredictable behaviour, in addition to more realistic hardware models and kernel functionality, to future work.

Finally we emphasize that virtualization and/or separation kernels are not the only tools available for secure partitioning. The ARM-proprietary TrustZone solution [11] adds to the standard ARMv7 architecture a secure partition that can be used to split the CPU resources between an untrusted and a trusted OS. Our results show that the kernel can be protected using standard, less expensive hardware, and a smaller TCB. Moreover, extending the proposed verification strategy can be straightforwardly extended to manage a different number (>2) of partitions.

[3]This is the element of our "real system" that is not really real.

Paper B

Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties

Narges Khakpour, Oliver Schwarz, and Mads Dam

Abstract

In this paper, we formally verify security properties of the ARMv7 Instruction Set Architecture (ISA) for user mode executions. To obtain guarantees that arbitrary (and unknown) user processes are able to run isolated from privileged software and other user processes, instruction level noninterference and integrity properties are provided, along with proofs that transitions to privileged modes can only occur in a controlled manner. This work establishes a main requirement for operating system and hypervisor verification, as demonstrated for the PROSPER separation kernel. The proof is performed in the HOL4 theorem prover, taking the Cambridge model of ARM as basis. To this end, a proof tool has been developed, which assists the verification of relational state predicates semi-automatically.

B.1 Introduction

The ability to execute application software in a manner which is isolated from other application software running on a shared processing platform is an essential prerequisite for security. This allows user applications or virtual machines to coexist without violating confidentiality or integrity of critical data, it allows critical system resources to be protected from user manipulation, it can help to prevent fault propagation, and it can be used to save costly hardware that might otherwise be needed to provide physical separation.

Isolation is typically provided by a mix of hardware and software. A memory management unit (MMU) may be used to provide basic memory protection, and the processor may be equipped with multiple privilege levels, running application programs as userland processes and kernel routines at privileged levels, with additional abilities to access and configure critical parts of the processor, the MMU, and various storage/display/peripheral devices attached to the processor.

In such a setting, isolation is a result of the correct interplay between hardware and kernel. It is the responsibility of the kernel to correctly manipulate the processor state to achieve the desired effects, whatever they may be (context switching, logging, fault management, device management, etc). It is the responsibility of the processing hardware to correctly implement the partitioning safeguards and mode transition conventions assumed by the kernel. For security, the kernel and the processor must both be correct and agree on their mode of interaction. Most formal kernel analyses in the literature [25, 90, 105, 125, 142] address the kernel software itself, in source or binary form, and leave the properties of the instruction set architecture (ISA) to be handled by fiat. Our contribution is to suggest a possible approach, including tool support, for performing the ISA specific security analysis, specifically for user mode execution.

We have identified two main concerns.

First, an implicit contract must exist which stipulates the "region of influence/dependency" of userland processes. That is, in a given user mode processor/MMU configuration it must be determined which memory locations and (control) registers can be read or written, or, in a more fine grained analysis, how information is able to flow to or from specific parts of the processor and the memory. User processes must be constrained in accessing or otherwise being influenced by critical resources of the kernel or of other user processes. This is not trivial. For instance, as shown by Duflot et al. [62], on some x86 processors it is possible for low-privilege code to overwrite higher privilege code by writing to an address that usually refers to the video card. To enable this attack, it suffices to first flip a configuration bit usually accessible from the low privilege level.

Second, kernel code relies on a set of mode switching conventions, for instance on ARM that program status registers and relevant user registers (including the program counter) are properly banked, the program counter is updated to point at the correct location in the vector table, and so on. If these conventions are not established by the processor and adhered to by the kernel, it may be possible for userland processes to induce various sorts of malicious behavior, for instance by letting a handler's link register point to a foreign address.

Performing this analysis is not trivial, particularly not if information flow is to be taken into account, as is done in this paper. All instructions, error conditions, and user to privileged mode transitions must be considered. The number of instructions is high and in modern processors a single instruction can involve a large number (order of 20-30) of atomic register or memory accesses.

In this paper, we identify and prove several partitioning-related properties of the ARMv7 ISA specification [12, 13] addressing user mode execution and mode switching. The first is an instruction level noninterference property related to the non-infiltration property in [90] stating that the behavior of an ARMv7 processor in user mode only depends on its accessible resources, mostly user registers, MMU configurations and the memory allocated to that process. The second, corresponding to the non-exfiltration property of [90], is an integrity property stating that, again while in user mode, the processor is unable to modify protected resources. A third set of properties concerns mode switching conventions. These properties have been applied in the PROSPER project [139] to verify isolation for the PROSPER separation kernel [53]. The PROSPER project aims at producing and verifying a fully functional secure hypervisor for embedded systems, providing services such as guest isolation, so that only explicitly allowed communication occurs.

Our proof uses the HOL4 [95] model of ARM, developed at Cambridge by Fox et al. [70]. We extend this model by simple memory protection. The ARMv7 ISA properties outlined above are formalized and proved. To make the quite sizable proof task feasible, we have developed a helper tool based on relational Hoare logic, that is able to automate significant parts of the proof.

To the best of our knowledge our work represents the first formalized analysis of the ARMv7 ISA. Others, specifically the Cambridge HOL4 group, have developed various helper tools for assembling, disassembling, executing, and managing ARM machine code and the HOL4 ARM ISA model [70, 126]. Also, the HOL4 ARM model has been used in several verification exercises in the literature, on software fault isolation (SFI) [186] and on the extension of the seL4 verification work [105] from C to binary level [156]. However, we have not yet seen general correctness properties formalized and verified for ARM at the ISA level. In fact, we believe the type of analysis presented here can be useful beyond kernel verification. For instance, formalized security properties can be useful to both improve the usefulness and precision of ISA specifications, and to enable developers obtain a concise description of secure configurations, without manual consideration of extensive architecture specifications.

B.2 The Formal Specification of ARM

We use Fox et al's monadic HOL4 model [70] of the ARMv7 ISA. This model covers the ARM, Thumb and ThumbEE instruction sets, comprising 81 instructions for branching, memory access, data processing, co-processor access, status access, and miscellaneous functionality. Figure B.1 shows a simplified definition of an ARM state in this model. The function `psrs` returns the value of a processor state register (of type `ARMpsr`). The processor state registers include the current program status register, `CPSR`, in addition to the banked psrs `SPSR_m` for each privileged mode `m`, except for system mode. Program status registers encode arithmetic flags, the processor mode `M`, interrupt masks (`I` for ordinary and `F` for fast interrupts) and instruction encoding. The ARMv7 core provides seven processor modes: one non-privileged user mode `usr`, and six privileged modes (`abt,fiq,irq,svc,und,sys`), activated when an exception (such as an interrupt) is invoked. Variants with the TrustZone extension [11] also have a monitor mode. However, this has to be invoked

```
arm-state = <| psrs         : PSRName -> ARMpsr;
               regs         : RName -> word32;
               memory       : word32 -> word8;
               coproc       : coprocessors;
               accesses     : memory_access list;
               misc         : Monitors # ARMinfo # bool # bool |>;
```

Figure B.1: The ARM state in HOL4

from a privileged mode and we consider its usage out of scope of this paper.

The function `regs` takes a register name and returns its value. The ARM registers include sixteen general purpose registers (`r0-r15`) that are available from all modes in addition to the banked registers of each privileged mode (except of `sys`) that are available only in that mode. Among the user registers, register `r13` functions as stack pointer `SP`, register `r14` as link register `LR` and register `r15` as program counter `PC`.

The function `memory` reads a byte (`word8`) from an address (`word32`). The field `coproc` represents those coprocessor registers in `CP14` and `CP15` that implicitly influence execution. The coprocessor registers central for this work are registers `SCTLR` , `TTBR0` and `DACR` of coprocessor 15. They, together with the page table, are used to configure the MMU. The field `misc` represents the exclusive monitors used for synchronization purposes, general information about the state, e.g. the architecture version, if the system is waiting for an interrupt etc, and `accesses` records the accesses to the memory.

A *computation* in the monadic HOL4 ARM model is a term of the following (slightly beautified) type

$$\alpha\ \texttt{M} = \texttt{arm\_state} \mapsto (\alpha, \texttt{arm\_state})\ \texttt{error\_option}.$$

where `error_option` is a datatype defined as follows:

```
(α,β) error_option = ValueState of α => β
                   | Error of string
```

Computations act on a state `arm_state` and return either `ValueState` $a\ s$, a new state s of type `arm_state` along with a return value a of type α, or an error e. The unpredictable computations, i.e., those that are underspecified by the ARM specification return an error. The monad unit `constT` injects a value into a computation, i.e. `constT` $a\ s$ = `ValueState` $a\ s$, while binding is a sequential composition operation

$$\begin{aligned} f_1 \gg\!=_e f_2 \quad = \quad & \lambda s.\texttt{case}\ f_1 s\ \texttt{of Error}\ c \rightarrow \texttt{Error}\ c \\ & \qquad\qquad || \ \texttt{ValueState}\ a\ s' \rightarrow \\ & \qquad\qquad\qquad \texttt{if}\ e\ s'\ \texttt{then}\ f_2\ a\ s'\ \texttt{else}\ f_1\ s. \end{aligned}$$

$\texttt{errorT}\ a = \texttt{Error}\ \ a$
$\texttt{condT}\ e\ f = \texttt{if}\ e\ \texttt{then}\ f\ \texttt{else}\ \texttt{constT}\ ()$
$\texttt{if}\ e\ \texttt{then}\ f_1\ \texttt{else} f_2 = \lambda s.\texttt{if}\ e\ s\ \texttt{then}\ f_1\ s\ \texttt{else}\ f_2\ s$
$f_1\ |||_e\ f_2 = f_1 \ggg\!=_e (\lambda x.f_2 \ggg\!=_e (\lambda y.\texttt{constT}\ (x, y)))$
$\texttt{forT}_e\ l\ h\ f = \texttt{if}\ l > h\ \texttt{then}\ \texttt{constT}\ []$
$\qquad\qquad\qquad \texttt{else}\ ((f\ l) \ggg\!=_e (\lambda r.\texttt{forT}_e\ (l+1)\ h\ f \ggg\!=_e (\lambda l.\texttt{constT}\ r :: l)))$

Figure B.2: Auxiliary monad operations

That is, if e holds in the final state of f_1, the return value of f_1 is passed to f_2 as the input parameter, otherwise f_2 is not executed.

In addition to unit and binding, the ARM monadic specification uses standard constructs for lambda, let, and cases, as well as the monad operations parallel composition ($f_1\ |||_e\ f_2$), positive conditional ($\texttt{condT}\ e\ f$), full conditional ($\texttt{if}\ e\ \texttt{then}\ f_1\ \texttt{else}\ f_2$), error ($\texttt{errorT}\ a$), and an iterator ($\texttt{forT}_e\ l\ h\ f$), (inductively) defined in Figure B.2.

B.3 Memory Management

The Memory Management Unit (MMU) enforces memory access policies and is therefore important for isolation. MMU configurations consist of page tables in memory and dedicated registers of `CP15`. Specific to ARM is the possibility of partitioning pages into collections of memory regions, so-called *domains*. The theorems in this paper are based on the concrete MMU configurations (memory ranges, the page table setup etc.) used in the PROSPER kernel. The coprocessor registers involved are `SCTLR`, `TTBR0` and `DACR`. The `SCTLR` register determines whether the MMU is enabled, `TTBR0` contains the base address of the page table, and `DACR` manages the ARM domains.

MMU Extension The evaluation function `permitted` takes as parameters a byte address, a flag indicating whether reading or writing access is to be evaluated, the values of `SCTLR`, `TTBR0` and `DACR`, a flag indicating whether permissions are to be checked against a privileged mode, and the memory containing the page tables. The pair of booleans returned by `permitted` states whether the access permission on the specified byte is defined in the given configuration and the outcome of that decision (`true` if access is granted). The PROSPER kernel uses a basic version of `permitted`, supporting one-level page tables without address translation, but including the interpretation of ARM domains. It is shown that `permitted` is defined for all addresses in all reachable states.

The history of memory accesses is tracked in the `accesses` field of the machine state, allowing to compute the set of memory pages accessed by an instruction.

```
next irpt s =
(clear_alist $\gg=_{\text{nav}}$
 ($\lambda$u. if irpt = NoInterrupt then
        waiting_for_interrupt $\gg=_{\text{nav}}$
        ($\lambda$wfi. condT ($\neg$wfi)
                  (fetch_instruction $\gg=_{\text{T}}$
                  ($\lambda$(opc, ins). is_viol $\gg=_{\text{T}}$ ($\lambda$av. clear_alist $\gg=_{\text{nav}}$
                  ($\lambda$u. if av then prefetch_abort
                       else
                         (execute ins $\gg=_{\text{T}}$ ($\lambda$u. is_viol $\gg=_{\text{T}}$
                         ($\lambda$av. condT av
                                (clear_alist $\gg=_{\text{nav}}$
                                ($\lambda$u. data_abort)))))))))
      else take_exception irpt $\gg=_{\text{nav}}$ ($\lambda$u. clear_wait_for_irpt))) s
```

Figure B.3: The `next` computation.

To stop computation after the first access violation, $\gg=_{\text{nav}}$ has been chosen as standard binding operator, where `nav s` ("no access violation") is `true` if and only if there is no entry in the access list of machine state `s` that causes `permitted` to return a negative answer int the current configuration of `s`. The recording of an access always happens before the access itself.

The instruction execution function `next` (see Figure B.3) takes an exception/interrupt flag `irpt` and a state `s` and produces the consequent state, by either initiating the demanded exception or by fetching and executing the next instruction pointed to by the `PC` in `s`. If an access violation is recorded after instruction fetching or execution, a prefetch or data abort exception (respectively) is initiated. The access list is cleared between the single steps, preventing the execution from halting and instead proceeding with exception handling. Occasionally, the unconditional binding $\gg=_{\text{T}}$ is used.

MMU Configuration Let `accessible i a` express that address `a` is readable and writable by user process `i`. The predicate `mmu_setup i s` holds if and only if (i) state `s` implements the desired access policy for process `i`, (ii) no MMU configuration for any address is underspecified, and (iii) none of the active page tables in `s` (represented by the address set `page_table_adds s`) is accessible according to the policy.

```
mmu_setup i s = ∀add, is_write, u, p.
   (u,p) = permitted add is_write (mmu_registers s) F s.memory
 ⇒ u ∧ ((accessible a i) ⇔ p)
      ∧ (a ∈ (page_table_adds s) ⇒ ¬(accessible a i))
```

B.4 Security Properties

We next turn to formalizing the instruction level partitioning properties. For user mode execution we formulate the requirements in terms of non-infiltration and non-exfiltration properties (cf. [90]), adapted to our setting.

Our model does not include caches, timing or hardware extensions such as TrustZone or virtualization support. Devices are not part of the model either; however, interrupts and other exceptions are taken into account, apart from fast interrupts and resets. Accordingly, the `fiq` and `mon` modes are outside of our analysis. As discussed, the chosen memory configuration is specific to the PROSPER project. Consequences of a limited coprocessor model and underspecified instructions are discussed in Section B.8.

B.4.1 Non-infiltration

Confidentiality of the kernel and neighboring user processes is guaranteed by non-infiltration, a noninterference-like property at the user mode single instruction level. Consider two machine states in user mode that are *low equivalent* in the sense that the two states agree on the resources (registers and memory locations) that are permitted to influence user mode execution, but do not necessarily agree on other resources. Non-infiltration holds if the poststates, after execution of one instruction, remain low equivalent (or produce the same error).

Theorem 4. Non-infiltration

```
∀s1, s2, i, irpt.  mode s1 = mode s2 = usr ∧ bisim i s1 s2
⇒ (∃t1, t2. next irpt s1 = ValueState () t1
          ∧  next irpt s2 = ValueState () t2 ∧ bisim i t1 t2)
∨  (∃e. next irpt s1 = Error e ∧ next irpt s2 = Error e)
```

The relation `bisim` is the low equivalence relation. User mode processes are allowed to be influenced by the user mode registers, the memory assigned to them, the `CPSR`, the coprocessors, pending access violations and the `misc` state component. Exclusive monitors (as field of `misc`) can inherently influence and be influenced by user mode software and need thus to be cleared by kernels on context switches.

```
bisim i s1 s2 =
    mmu_setup i s1 ∧ mmu_setup i s2 ∧ (equal_user_regs s1 s2)
  ∧ (∀a. (accessible i a) ⇒ (s1.memory a = s2.memory a))
  ∧ (s1.psrs(CPSR)= s2.psrs(CPSR)) ∧ (s1.coproc.state = s2.coproc.state)
  ∧ (nav s1 = nav s2) ∧ (s1.misc = s2.misc)
  ∧ s1.psrs(spsr_(mode s1)) = s2.psrs(spsr_(mode s2))
  ∧ s1.regs(lr_(mode s1)) = s2.regs(lr_(mode s2))
```

The two last items have been included to assure that `SPSR` and link register (of a possibly privileged poststate) only depend on resources allowed to influence user mode execution as well, so that they can actually be restored later on.

B.4.2 Non-exfiltration

Non-exfiltration guarantees the integrity of resources foreign to the active user process. It expresses that, given an MMU setup for user process `i` active, the execution of a single instruction in user mode will not modify any other resources but those considered to be modifiable by `i`.

Theorem 5. Non-exfiltration

```
∀s, t, i, irpt. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ⇒ unmodified i s t
```

Here, `unmodified` expresses the desired relation between the prestate `s` and the poststate `t` of an active process `i`. We require that coprocessors, the fast interrupt flag and any memory not belonging to `i` remain unchanged. The only registers allowed to change are the `CPSR`, the user mode registers, and the `PSR` and the link register of the mode in `t`. The interrupt flag of the `CPSR` is not modified when staying in user mode.

```
unmodified i s t =
     (s.coproc = t.coproc) ∧ (s.psrs(CPSR).F = t.psrs(CPSR).F)
   ∧ (∀a. ¬(accessible i a) ⇒ (s.memory a = t.memory a))
   ∧ ((mode s ∈ {usr, mode t} ∧ mode t ∈ {usr, fiq, irq, svc, abt, und})
    ⇒( (∀reg. reg ∉ accessible_regs(mode t) ⇒ s.regs(reg) = t.regs(reg))
      ∧ (∀psr. psr ∉ {CPSR, spsr_(mode t)} ⇒ s.psrs(psr) = t.psrs(psr))
      ∧ (mode t = usr ⇒((s.psrs(CPSR)).I = (t.psrs(CPSR)).I))))
```

B.4.3 Switching to Privileged Modes

Secure user mode execution is not by itself sufficient. It is also necessary to consider transitions to privileged modes to prevent user processes from privileged execution rights. No user process should be able to effect a mode change with the `PC` set to a memory location of his choice. Instead, all entry points into privileged modes should be in the exception vector table. Similarly, even though user processes are allowed to choose a different endianness for their own execution, that should not influence the interpretation of the system handlers when switching back to privileged mode. Theorem 6 covers those additional constraints.

Theorem 6. Privileged Constraints

```
∀s, t, i, irpt. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ⇒ priv_const s t
```

Besides the above properties, the relation `priv_const` lists the reachable processor modes[1] and assures that interrupts are masked when entering a privileged

[1]Monitor and system mode can only be reached from another privileged mode.

mode. Also, status register flags regarded as unwritable will be copied from the CPSR in prestate s to the SPSR in poststate t. This guarantees that a kernel can restore the saved program status register without further modifications when jumping back to the user process. Otherwise, user processes would be able to make the kernel enable/disable interrupts or change their execution mode. All access violations, if there were any, will have been handled (nav t).

```
priv_const s t =
  mode t ∈  {usr, fiq, irq, svc, abt, und}
  ∧ (mode t ≠ usr ⇒
     (   t.regs(PC) ∈ vt_adds(vt_base s, mode t) ∧ nav t
      ∧ (t.psrs(CPSR)).(I, J, IT, E) = (T, F, 0w, endianess s)
      ∧ (t.psrs(spsr_(mode t))).(M, I, F)
          = (usr, (s.psrs(CPSR)).I, (s.psrs(CPSR)).F)))
```

B.4.4 Link Register Contents in Supervisor Mode

Upon reception of a software interrupt, exception handlers in the invoked supervisor mode (svc) often need to analyze the calling instruction, in order to determine the software interrupt number for example. Therefore, verification might require assertions that the memory location pointed to by the link register actually does belong to the user process which caused the switch to supervisor mode. Formally, when going from state s in user mode to state t in supervisor mode, it is required that the svc-link register of t (i) is equal to the PC of s plus an instruction set dependent offset and (ii) corrected by the offset, points to an aligned word that is readable in t (independent of the mode). Note that offset and width of the word depend on the instruction set used by the user process, not on the one used by the handler.

Theorem 7. Link Register Constraints

```
∀s, t, i, irpt, lr. mode s = usr ∧ mmu_setup i s
  ∧ next irpt s = ValueState () t ∧ mode t = svc ∧ lr = t.regs(LR_svc)
⇒ lr = s.regs(PC) + offset s
  ∧ ((t.psrs(SPSR_svc)).T ⇒ aligned_word_readable t T (lr - 2w))
  ∧ (¬(t.psrs(SPSR_svc)).T ∧ ¬(t.psrs(SPSR_svc)).J
          ⇒ aligned_word_readable t F (lr - 4w))
```

Here, aligned_word_readable s b add states that the aligned word referred to by add is readable in s. Dependent on whether b is true or false, word width and alignment are 16 or 32 bit.

B.4.5 Safe User Mode Execution

The final aim is to guarantee that as long as the machine is executing in user mode, it causes no noninterference or integrity violations. Let $s_1 \rightsquigarrow s_n$ denote a sequence of next computations $s_1 \rightarrow s_2 \rightarrow .... \rightarrow s_n$ in user mode, i.e. mode $s_i = \texttt{usr}$, $1 \leq i < n$ and mode $s_n \neq \texttt{usr}$. The following theorem assures the safe execution and safe mode switching of a user process.

Theorem 8. Let $s_1 \rightsquigarrow s_n$ and mmu_setup i s_1, (i) if $s'_1 \rightsquigarrow s'_n$ and bisim i s_1 s'_1 then bisim i s_n s'_n, (ii) unmodified i s_1 s_n, and (iii) priv_const s_{n-1} s_n.

The proof of (i) and (ii) is an easy induction on n using theorems 4 and 5. Item (iii) follows from Theorem 6.

B.5 The Logic Framework

Considering the size and complexity of the ARM model and the instruction set, to prove the properties of the previous section tool support is essential. In this section we present proof rules for relational and invariant reasoning that help to automate the proof.

Non-infiltration The proof uses a relational Hoare logic based on assertions {f:R →R'} defined as follows:

$$
\begin{aligned}
\{f\texttt{:R} \rightarrow \texttt{R'}\} = \forall s_1,s_2.\ \texttt{R}\ s_1\ s_2 \Rightarrow\ & \\
&(\exists a,t_1,t_2.\ f\ s_1 = \texttt{ValueState}\ a\ t_1 \wedge \\
&\qquad f\ s_2 = \texttt{ValueState}\ a\ t_2 \wedge \texttt{R'}\ t_1\ t_2) \\
&\vee(\exists e. f\ s_1 = \texttt{Error}\ e \wedge f\ s_2 = \texttt{Error}\ e)
\end{aligned}
$$

The judgment asserts that, if started in prestates s_1, s_2 related by prerelation R, either the executions of the monadic computation f return identical values a with poststates t_1, t_2 related by postrelation R', or else they both return the same error e.

For the analysis it suffices to consider a fixed set of relations

$$\texttt{R\_m} = \lambda s_1.\lambda s_2.\texttt{bisim i}\ s_1\ s_2 \wedge \texttt{mode}\ s_1 = \texttt{m} \wedge \texttt{mode}\ s_2 = \texttt{m}$$

or R_(n,m) = R_n ∪ R_m.

Figure B.4 shows the relational logic inference rules. The inference system is incomplete, but sufficient for our purpose. A relation R_m is preserved by errorT and constT (rules constTR and errorTR), and if a computation preserves one of the R_m relations then that computation can be used in a conditional or a *for* loop as well (condTR, conR and forTR). The rule widenR and absR are used to weaken the postrelation and reason about lambda computations, respectively. The rule seqTR states that the postrelation of $f \gg=_{\texttt{nav}} f'$ is the union of the postrelations of f and f', provided that either f preserves R_n or f' preserves R_k. If there is an access violation after f, the computation stops and R_n must hold. Otherwise, f' will execute and R_k must hold. Thus, the postrelation is the union of R_n and R_k.

Theorem 9. All assertions $\{f : R \rightarrow R'\}$ derivable according to the inference rules in Figure B.4 are valid.

$$\text{errorTR}\,\frac{}{\{\texttt{errorT}\ a : \texttt{R\_m} \to \texttt{R\_m}\}} \qquad \text{constTR}\,\frac{}{\{\texttt{constT}\ a : \texttt{R\_m} \to \texttt{R\_m}\}}$$

$$\text{condTR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_m}\}}{\{\texttt{condT}\ \psi\ f : \texttt{R\_m} \to \texttt{R\_m}\}} \qquad \text{forTR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_m}\}}{\{\texttt{forT}_{\texttt{nav}}\ l\ h\ f : \texttt{R\_m} \to \texttt{R\_m}\}}$$

$$\text{conR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_m} \to \texttt{R\_n}\}}{\{\texttt{if}\ \psi\ \texttt{then}\ f\ \texttt{else}\ f' : \texttt{R\_m} \to \texttt{R\_n}\}}$$

$$\text{widenR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\}}{\{f : \texttt{R\_m} \to \texttt{R\_(n,k)}\}} \qquad \text{absR}\,\frac{\forall y.\{f\ y : \texttt{R\_m} \to \texttt{R\_n}\}}{\{\lambda y.f : \texttt{R\_m} \to \texttt{R\_n}\}}$$

$$\text{seqTR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_n} \to \texttt{R\_k}\} \quad (\texttt{m} = \texttt{n}) \vee (\texttt{n} = \texttt{k})}{\{f \gg\!=_{\texttt{nav}} f' : \texttt{R\_m} \to \texttt{R\_(n,k)}\}}$$

$$\text{parTR}\,\frac{\{f : \texttt{R\_m} \to \texttt{R\_n}\} \quad \{f' : \texttt{R\_n} \to \texttt{R\_k}\} \quad (\texttt{m} = \texttt{n}) \vee (\texttt{n} = \texttt{k})}{\{f|||_{\texttt{nav}}f' : \texttt{R\_m} \to \texttt{R\_(n,k)}\}}$$

Figure B.4: Relational inference rules

$$\text{errorTI}\,\frac{}{\texttt{INV}\langle \texttt{errorT}\ a, \texttt{Q}, \texttt{P}\rangle} \qquad \text{constTI}\,\frac{\texttt{refl P}}{\texttt{INV}\langle \texttt{constT}\ c, \texttt{Q}, \texttt{P}\rangle}$$

$$\text{condTI}\,\frac{\texttt{refl P} \ \ \texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle}{\texttt{INV}\langle \texttt{condT}\ e\ f, \texttt{Q}, \texttt{P}\rangle} \qquad \text{forTI}\,\frac{\texttt{refl P} \quad \texttt{trans P} \quad \texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle}{\texttt{INV}\langle \texttt{forT}_e\ l\ h\ f, \texttt{Q}, \texttt{P}\rangle}$$

$$\text{conRI}\,\frac{\texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle \quad \texttt{INV}\langle f', \texttt{Q}, \texttt{P}\rangle}{\texttt{INV}\langle \texttt{if}\ \psi\ \texttt{then}\ f\ \texttt{else}\ f', \texttt{Q}, \texttt{P}\rangle}$$

$$\text{absI}\,\frac{\forall y.\texttt{INV}\langle f\ y, \texttt{Q}, \texttt{P}\rangle}{\texttt{INV}\langle \lambda y.f, \texttt{Q}, \texttt{P}\rangle} \qquad \text{seqTI}\,\frac{\texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle \quad \texttt{INV}\langle f', \texttt{Q}, \texttt{P}\rangle \quad \texttt{trans P}}{\texttt{INV}\langle f \gg\!=_e f', \texttt{Q}, \texttt{P}\rangle}$$

$$\text{parTI}\,\frac{\texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle \quad \texttt{INV}\langle f', \texttt{Q}, \texttt{P}\rangle \quad \texttt{trans P}}{\texttt{INV}\langle f|||_e f', \texttt{Q}, \texttt{P}\rangle}$$

Figure B.5: Invariant inference rules

Non-exfiltration Similar to the non-infiltration proof, the proof of non-exfiltration uses a sound but incomplete inference system, this time concerning computation invariants of the following shape:

$$\texttt{INV}\langle f, \texttt{Q}, \texttt{P}\rangle = \forall s, t.\ \texttt{Q}\ s\ \wedge f\ s = \texttt{ValueState}\ a\ t \implies \texttt{P}\ s\ t\ \wedge\ \texttt{Q}\ t\ .$$

That is, if Q holds of the prestate then `P` holds of the prestate-poststate pair, and `Q` of the poststate. We use a simple collection of inference rules to prove `Q` and `P` , shown in Figure B.5. In this figure, `refl P` and `trans P` respectively state that `P` is reflexive and transitive. For non-exfiltration we need to prove that `unmodified i` is satisfied during the execution of each instruction both when it ends in user mode

```
take_svc_exception  = IT_advance  ≫=nav
   (λ u.(read_reg 15w |||nav exc_vector_base |||nav read_cpsr |||nav
               read_scr |||nav read_sctlr )≫=nav
     (λ(pc,ExcVectorBase,cr,scr,sctlr).
       (condT (cr.M = 0b10110w) (write_scr  (scr with NS := F))  |||nav
        write_cpsr (cr with M := 0b10011w)) ≫=nav
        (λ (u1,u2). (write_spsr cr |||nav
          write_reg 14w (if cr.T then pc - 2w else pc - 4w) |||nav
          (read_cpsr ≫=nav
           (λ cr'.write_cpsr (cr' with
                             <| I := T; IT := 0b00000000w;J := F;
                                T := sctlr.TE; E := sctlr.EE |>))) |||nav
            branch_to (ExcVectorBase + 8w)) ≫=nav unit4)))
```

Figure B.6: The HOL4 code for switching to svc mode [95]

and when switching to privileged mode. A prerequisite for this is that the MMU is configured correctly during computation. To prove the non-exfiltration property, we check $\mathtt{INV}\langle \mathtt{next}, \mathtt{mmu\_setup\ i}, \mathtt{unmodified\ i}\ \rangle$.

Theorem 10. All assertions $\mathtt{INV}\langle f,\mathtt{Q},\mathtt{P}\rangle$ derivable according to the inference rules in Figure B.5 are valid.

Privileged Constraints The final goal is to prove that next establishes the relation priv_const, a conjunction of primitive constraints P. Since the primitive constraints do not always hold during computations in privileged mode, the inference rules of Figure B.5 are generally not able to prove this property. To make verification tractable, we prove primitive constraints locally at the point in the monadic computation where it is established and then use a set of inference rules to infer its correctness for the entire computation. We illustrate the proof using an example. In the ARM model, all computations which lead to a privileged mode m end by a computation called take_m_exception. Figure B.6 shows the function take_svc_exception for switching to supervisor mode. Let this computation start in state s1 and end in state sn. Consider the primitive constraint $\mathtt{P}_{\mathtt{psr}}$ stating that SPSR_svc of the final state sn must be equal to CPSR of the initial state s1. Let t and $\mathtt{t}'$, respectively be the initial state and final state of write_spsr cr and m be the mode of $\mathtt{t}'$. The computation write_spsr cr writes the value of free variable cr into SPSR_m and establishes the property $\mathtt{P}'_{\mathtt{psr}} \overset{\text{def}}{=} \mathtt{t}'.\mathtt{psrs}(\mathtt{SPSR\_m}) = \mathtt{cr}$. We call write_spsr cr a $\mathtt{P}'_{\mathtt{psr}}$-establisher. A computation g is P-establisher, if independently of its input state, P holds in its output state, i.e.

$$\mathtt{P{-}establ}(g) = \forall s, a, t.\ g\ s = \mathtt{ValueState}\ a\ t\ \wedge\ \mathtt{nav}\ t\ \Longrightarrow\ \mathtt{P}\ t$$

We can prove that the block starting from write_spsr cr establishes $\mathtt{P}'_{\mathtt{psr}}$ as well, because the rest of the computations of this block does not modify this property. Then we can prove that the free variable cr takes the value s1.psrs(CPSR), and m

$$\text{seqTS1}\,\frac{\texttt{P-establ}(f) \qquad \texttt{INV}\langle f', \texttt{P}, \top\rangle}{\texttt{P-establ}(f \gg\!=_{\texttt{nav}} f')} \qquad \text{seqTS2}\,\frac{\texttt{P-establ}(f)}{\texttt{P-establ}(f' \gg\!=_{\texttt{nav}} f)}$$

$$\text{parTS1}\,\frac{\texttt{P-establ}(f) \qquad \texttt{INV}\langle f', \texttt{P}, \top\rangle}{\texttt{P-establ}(f \mathbin{|||_{\texttt{nav}}} f')} \qquad \text{parTS2}\,\frac{\texttt{P-establ}(f)}{\texttt{P-establ}(f' \mathbin{|||_{\texttt{nav}}} f)}$$

$$\text{absS}\,\frac{\forall y.\texttt{P-establ}(f\ y)}{\texttt{P-establ}(\lambda y.f)}$$

Figure B.7: Privileged constraints inference rules

is bound to `svc`. Thus, `sn.psrs(SPSR_svc)` = `s1.psrs(CPSR)` holds for the computation block from `write_spsr cr`. As this block is a $\texttt{P}_{\texttt{psr}}$-establisher, we conclude that the computations before `write_spsr` do not influence the established property and $\texttt{P}_{\texttt{psr}}$ is satisfied by `take_svc_exception`.

Figure B.7 shows the `P`-establisher inference rules. These rules along with the inference rules of Figure B.5 are used to prove the privileged constraints. The rule seqTS1 states that if the monadic computation f is a `P`-establisher and `P` is an invariant of f', then the sequential composition $f \gg\!=_{\texttt{nav}} f'$ is `P`-establisher. The rule seqTS2 describes that if the monadic computation f is a `P`-establisher, then $f' \gg\!=_{\texttt{nav}} f$ is also `P`-establisher. Similar rules are defined for the $|||_{\texttt{nav}}$ operator.

Theorem 11. All assertions `P-establ`(f) derivable according to the inference rules in Figure B.7 are valid.

B.6 Implementation and Evaluation

Implementation We use the HOL4 theorem prover to verify our properties. The central assets of our work are available from [139]. We have developed a tool, ARM-prover, to automate the verification process based on the proof systems in Fig. B.4 and B.5. To avoid having to explore the instruction set more than once the prover actually combines the theorems 4, 5 and 6 into one.

The proof systems do not provide rules for `case` and `let` statements. These are easily handled using standard HOL4 simplification. Other monadic expressions are refined using the inference rules in Fig. B.4 and B.5 in a top down fashion. The proofs for “write” primitives as well as register and memory accesses in user mode are done manually, but the tool can handle some of the “read” computations directly, allowing to prove a large share of the workload automatically.

A particular difficulty concerns binding. When a binding expression $f_1 \gg\!=_{\texttt{nav}} f_2$ is decomposed the return value of f_1 becomes unbound in f_2. To handle this we simplify computations by embedding more information before calling the prover, using some auxiliary lemmas. For example, the following formula states that `cpsr` in computation `H` following `read_cpsr` can be substituted by the CPSR in prestate `s` with mode `m`.

```
(mode s = m) ⇒ (read_cpsr ≫=nav (λcpsr. H(cpsr))) s =
        (read_cpsr ≫=nav (λcpsr. H(s.psrs(CPSR) with M:=m))) s
```

For the case that an instruction leads to a privileged mode, the last execution phase of the instruction, called switching phase, is in privileged mode. However, the privileged constraints first have to be established over the course of several steps and do not hold from the beginning. Since we can not use the ARM-prover tool to prove them automatically, we prove the privileged constraints for the switching phase manually.

Evaluation The Cambridge model of ARM is 9 kLOC. In addition to the ARM model, we rely mainly on the relatively small inference kernel of the HOL4 theorem prover, our MMU extension (about 180 lines of definitions) and the formulation of the discussed properties (about 290 lines). The entire proof script has a length of about 13 kLOC and needs roughly an hour to run on an Intel(R) Xeon(R) X3470 core. We invested about one person year of effort into this work.

B.7 Related Work

Several recent works address kernel verification. Some target information flow properties [25, 90, 125, 142], based on variants of noninterference [77]. Other work establishes a refinement relation between kernel code, in some representation, and an abstract specification. For the seL4 microkernel this was first performed for its C implementation [105] and is now extended to binary level [156]. As is the case with most refinement/simulation-based approaches, this work does not address information flow. In recent work on seL4 verification, Murray et al. [124, 125] present an unwinding-style characterization of intransitive noninterference. They introduce a proof calculus on nondeterministic state monads that is similar to that of this work. Their assertions are more general, however our proof rules cover several monadic operators and statements. In addition, we introduce rules to prove properties about executions that relate the final state of a computation to its initial state.

Alkassar et al. [5] describe the emulation of a simplified MIPS machine in C. The emulator allows the use of VCC to automatically check that every reachable state of a guest on a hypervisor is also reachable when the guest is running on a completely isolated machine. The C emulator has been adopted to verify parts of the hypervisor that mix C and assembly [134], and allows unknown user processes to be considered. Information flow properties are not considered, however.

Wilding et al. [180] formally proved exfiltration, infiltration and mediation theorems for the partitioning system of the AAMP7G microprocessor in ACL2. The hardware architecture differs from the one of ARM in several points, such as that there are no user-visible registers or that AAMP7G itself functions as a separation kernel. Proofs were performed using abstraction/refinement techniques and address kernel microcode. The verification led to a MILS certificate on Evaluation Assurance Level 7.

The ARMor system [186] sandboxes applications on ARM and provides formal verification of memory safety and control flow integrity, using the Cambridge HOL4 ARM model. Its software fault isolation does not use hardware features such as an MMU, but uses instead rewriting and subsequent verification of the compiled programs. This implies performance overhead, limitations on supported programs and verification processes in the extend of hours for each program. Furthermore, ARMor only establishes memory write protection; neither confidentiality nor protection of privileged registers is addressed.

Most works on kernel verification address handler code only and do not consider user mode execution. In a few cases [5, 146] user mode execution is considered, but without justification in terms of concrete processor access modalities. The main contribution of our work, over and beyond the above works, is that we attempt to justify the critical assumptions on processor level information flow in user mode execution through analysis at the level of a formalized ISA model.

Heitmeyer et al. [90] introduce non-exfiltration, non-infiltration, kernel integrity and data/control separation properties to verify a separation kernel. Since we focus on user-mode execution, those properties apply only partially here. Our non-infiltration property is the same as in [90], but the non-exfiltration property in our work covers both their kernel integrity and non-exfiltration.

B.8 Conclusion

We introduced and proved several security properties including a non-exfiltration, a non-infiltration and a safe switching property for user mode executions on the ARM architecture, using the Cambridge HOL4 ISA model. A logical framework based on (relational) Hoare logic has been developed for the analysis, supported by a tool, ARM-prover, which helps automate the proof. The ARM-prover can be used to prove general invariants about the ARM model (i.e., statements that need to hold at each execution point). We are planning to continue the development of the ARM-prover to improve automation further and cater for more general proof tasks.

Our results concerning register contents are generally valid and with small adaptations applicable in isolation verification of other hypervisors, separation kernels, and operating systems. Statements on memory safety depend on our specific setup. A reformulation that is independent of concrete MMU configurations should require a minor effort and is planned for future work.

The HOL4 model of ARM supports a partial coprocessor model. We made the assumption that the access to coprocessors via dedicated instructions is always denied in user mode. To have a more precise analysis and cover all possible side channels, a more comprehensive model of the available coprocessors involving all registers, the coprocessors' behavior and an acceptance/rejection-mechanism for register reads and writes that follows the specification is required. During context switches kernels need to mediate coprocessor registers user-accessible by dedicated coprocessor instructions. All other coprocessor registers are guaranteed to be non-modifiable in user mode. However, kernels must not introduce information flow from non-active processes to the coprocessor registers that are part of the present ARM model, since those might influence user mode execution.

Instructions that are underspecified ("unpredictable") in the ARM Architecture Reference Manual (ARMARM) are problematic. The ARM specification states that "*unpredictable* behavior must not perform any function that cannot be performed at the current or lower level of privilege using instructions that are not *unpredictable*"[13]. In one interpretation of this statement, theorems 5, 6 and 7 are valid on unpredictable instructions as well. In general, this is not true for non-infiltration. Yet, ARMARM requires further that "*unpredictable* behavior must not represent security holes" [12]. This formulation is very vague. However, we make the assumption that non-infiltration is preserved. In fact, we argue that the security properties we have presented provide manufacturers of ARM processors with a precise description of secure behavior for unpredictable cases.

Paper C

Automatic Derivation of Platform Noninterference Properties

Oliver Schwarz and Mads Dam

Abstract

For the verification of system software, information flow properties of the instruction set architecture (ISA) are essential. They show how information propagates through the processor, including sometimes opaque control registers. Thus, they can be used to guarantee that user processes cannot infer the state of privileged system components, such as secure partitions. Formal ISA models - for example for the HOL4 theorem prover - have been available for a number of years. However, little work has been published on the formal analysis of these models. In this paper, we present a general framework for proving information flow properties of a number of ISAs automatically, for example for ARM. The analysis is represented in HOL4 using a direct semantical embedding of noninterference, and does not use an explicit type system, in order to (i) minimize the trusted computing base, and to (ii) support a large degree of context-sensitivity, which is needed for the analysis. The framework determines automatically which system components are accessible at a given privilege level, guaranteeing both soundness and accuracy.

C.1 Introduction

From a security perspective, isolation of processes on lower privilege levels is one of the main tasks of system software. More and more vulnerabilities discovered in operating systems and hypervisors demonstrate that assurance of this isolation is far from given. That is why an increasing effort has been made to formally verify system software, with noticeable progress in recent years [90, 105, 124, 53, 5]. However, system software depends on hardware support to guarantee isolation. Usually, this involves at least the ability to execute code on different privilege levels and with basic memory protection. Kernels need to control access to their own code and data and to critical software, both in memory and as content of registers or other components. Moreover, they need to control the management of the access control itself. For the correct configuration of hardware, it is essential to understand how and under which circumstances information flows through the system. Hardware must comply to a contract that kernels can rely on. In practice, however, information flows can be indirect and hidden. For example, some processors automatically set control flags on context switches that can later be used by unprivileged code to see if neighbouring processes have been running or to establish a covert channel [161]. Such attacks can be addressed by the kernel, but to that end, kernel developers need machinery to identify the exact components available to unprivileged code, and specifications often fail to provide this information in a concise form. When analysing information flow, it is insufficient to focus on direct register and memory access. Confidentiality, in particular, can be broken in more subtle ways. Even if direct reads from a control flag are prevented by hardware, the flag can be set as an unintended side effect of an action by one process and later influence the behaviour of another process, allowing the latter to learn something about the control flow of the former.

In this paper we present a framework to automate information flow analysis of instruction set architectures (ISAs) and their operational semantics inside the interactive theorem prover HOL4 [95]. We employ the framework on ISA models developed by Fox et al. [68] and verify *noninterference*, that is, that secret (*high*) components can not influence public (*low*) components. Besides an ISA model, the input consists of desired conditions (such as a specific privilege mode) and a candidate labelling, specifying which system components are already to be considered as low (such as the program counter) and, implicitly, which components might possibly be high. The approach then iteratively refines the candidate labelling by downgrading new components from high to low until a proper noninterference labelling is obtained, reminiscent of [96]. The iteration may fail for decidability reasons. However, on successful termination, both soundness and accuracy are guaranteed unless a warning is given indicating that only an approximate, sound, but not necessarily accurate solution has been found.

What makes accurate ISA information flow analysis challenging is not only the size and complexity of modern instruction sets, but also particularities in semantics and representation of their models. For example, arithmetic operations (e.g., with bitmasks) can cancel out some information flows and data structures can contain a mix of high and low information. Modification of the models to suit the analysis is error prone and requires manual effort. Automatic, and provably correct, preprocessing of the specifications could overcome some, but not all, of those difficulties, but then the added value of standard approaches such as type systems over a direct implementation becomes questionable. By directly embedding noninterference into HOL4, we can make use of machinery to address the discussed difficulties and at the same time we are able to minimize the trusted computing base (TCB), since the models, the preprocessing and the actual reasoning are all implemented/represented in HOL4. Previous work on HOL4 noninterference proofs for ISA models [102] had to rely on some manual proofs, since its compositional approach suffered from the lack of sufficient context in some cases (e.g., the secrecy level of a register access in one step can depend on location lookups in earlier steps). In contrast, the approach suggested in the present paper analyses ISAs one instruction at a time, allowing for accuracy and automation at the same time. However, since many instructions involve a number of subroutines, this instruction-wide context introduces complexity challenges. We address those by unfolding definitions of transitions in such a way that their effects can be extracted in an efficient manner.

Our analysis is divided into three steps: (i) *rewriting* to unfold and simplify instruction definitions, (ii) the *actual proof attempt*, and (iii) automated counter-example-guided *refinement of the labelling* in cases where the proof fails. The framework can with minor adaptations be applied to arbitrary HOL4 ISA models. We present benchmarks for ARMv7 and MIPS. With a suitable labelling identified, the median verification time for one ARMv7 instruction is about 40 seconds. For MIPS, the complete analysis took slightly more than one hour and made configuration dependencies explicit that we had not been aware of before. We report on the following contributions: (i) a backward proof tactic to automatically verify noninterference of HOL4 state transition functions, as used in operational ISA semantics; (ii) the automated identification of sound and accurate labellings; (iii) benchmarks for the ISAs of ARMv7-A and MIPS, based on an SML-implementation of the approach.

C.2 Processor Models

C.2.1 ISA Models

In the recent years, Fox et al. have created ISA models for x86-64, MIPS, several versions of ARM and other architectures [70, 68]. The instruction sets are modelled based on official documentations and on the abstraction level of the programmer's view, thus being agnostic to internals like pipelines. The newest models are produced in the domain-specific language L3 [68] and can be exported to the interactive theorem prover HOL4. Our analysis targets those purely-functional HOL4 models for single-core systems. An ISA is formalized as a state transition system, with the machine state represented as record structure (on memory, registers, operational modes, control flags, etc.) and the operational semantics as functions (or *transitions*) on such states. The top-level transition `NEXT` processes the CPU by one instruction. While L3 also supports export to HOL4 definitions in monadic style, we focus our work on the standard functional representation based on let-expressions. States resulting from an *unpredictable* (i.e., underspecified) operation are tagged with an exception marker (see Section C.7 for a discussion).

C.2.2 Notation

A *state* $s = \{C_1 := c_1, C_2 := c_2, \ldots\}$ is a record, where the fields $C_1, C_2, \ldots$ depend on the concrete ISA. As a naming convention, we use R_i for fields that are records themselves (such as control registers) and F_i for fields of a function/mapping type (such as general purpose register sets). The *components* of a state are all its fields and subfields (in arbitrary depth), as well as the single entries of the state's mappings. The value of field C in s is derived by $s.C$. An update of field C in s with value c is represented as $s[C := c]$. Similarly, function updates of F in location l by value v are written as $F[l := v]$. Conditionals and other case distinctions are written as $\mathbb{C}(b, a_1, a_2, \ldots, a_k)$, with b being the selector and $a_1, a_2, \ldots, a_k$ the alternatives. A transition Φ transforms a pre-state s into a return-value v and a post-state s', formally $\Phi s = (v, s')$. Usually, a transition contains subtransitions $\Phi_1, \Phi_2, \ldots, \Phi_n$, composed of some structure ϕ of abstractions, function applications, case distinctions, sequential compositions and other semantic operators, so that $\Phi s = \phi(\Phi_1, \Phi_2, \ldots, \Phi_n)s$. Transition definitions can be recursively unfolded: $\phi(\Phi_1, \ldots, \Phi_n)s = \phi(\phi_1(\Phi_{1,1}, \ldots, \Phi_{1,m}), \ldots, \Phi_n)s = \ldots = \vec{\phi}s$, where $\vec{\phi}$ is the completely unfolded transition, called the *evaluated form*. For the transitions of the considered instruction sets, unfolding always terminates. Note that '=' is used here for the equivalence of states, transitions or values, not for the syntactical equivalence of terms. Below we give the definition of the ARMv7-NOOP-instruction and its evaluated (and simplified) form:

$$\begin{aligned} & \mathtt{dfn'NoOperation}\ s \\ = \ & \mathtt{BranchTo}(s.\mathtt{REG}\ \mathtt{RName\_PC}\ + \mathbb{C}(\mathtt{FST}\ (\mathtt{ThisInstrLength}\ ()\ s) = 16, 2, 4))\ s \\ = \ & ((), s[\mathtt{REG} := s.\mathtt{REG}[\mathtt{RName\_PC} := s.\mathtt{REG}\ \mathtt{RName\_PC}\ + \mathbb{C}(s.\mathtt{Encoding} = \mathtt{Thumb}, 2, 4)]]) \end{aligned}$$

NOOP branches to the current program counter ($s.$REG RName_PC) plus some offset. The offset depends on the current instruction length, which in turn depends on the current encoding. Here, FST selects the actual return value of the ThisInstrLength transition, ignoring its unchanged post-state.

C.2.3 Memory Management

For simplicity, our analysis focuses on core-internal flows (e.g., between registers) and abstracts away from the concrete behaviour of the memory subsystem (including address translation, memory protection, caching, peripherals, buses, etc.). Throughout the course of the - otherwise core internal - analysis, a contract on the memory subsystem is assumed that then allows the reasoning on global properties. The core can communicate with the memory subsystem through an interface, but never directly accesses its internal state. The interface expects inputs like the type of access (read, fetch, write, ...), the virtual address, the privilege state of the processor, and other parameters. It updates the state of the memory subsystem and returns a success or error message along with possibly read data. While being agnostic about the concrete behaviour of the memory subsystems, we assume that there is a secure memory configuration $\mathcal{P}_m$, restricting unprivileged accesses, e.g., through page table settings. Furthermore, we assume the existence of a low-equivalence relation $\mathcal{R}_m$ on pairs of memory subsystems. Typically, two memories in $\mathcal{R}_m$ would agree on memory content accessible in an unprivileged processor mode. When in unprivileged processor mode and starting from secure memory configurations, transitions on memory subsystems are assumed to maintain both the memory relation and secure configurations. Consider an update of state s assigning the sum of the values of register y and the memory at location a to register x, slightly simplified: $s[x := s.y + \texttt{read}(a, s.\texttt{mem})]$. Since `read` - as a function of the memory interface - satisfies the constraints above, for two pre-states s_1 and s_2 satisfying $\mathcal{P}_m s_1.\texttt{mem} \wedge \mathcal{P}_m s_2.\texttt{mem} \wedge \mathcal{R}_m(s_1.\texttt{mem}, s_2.\texttt{mem})$, we can infer that `read` will return the same value or error. Overall, with preconditions met, two states that agree on x, y, and the low parts of the memory before the computation, will also agree after the computation. That is, as long as `read` fulfils the contract, the analysis of the core (and in the end the global analysis) does not need to be concerned with details of the memory subsystem.

C.3 ISA Information Flow Analysis

C.3.1 Objectives

Consider an ISA model with an initial specification determining some preconditions (e.g., on the privilege mode) and some system components, typically only the program counter, that are to be regarded as observable (or *low*) by some given actor. If there is information flow from some other component (say, a control register) to some of these initially-low components, this other component must be regarded as observable too for noninterference to hold. The objective of the analysis is to identify all these other components that are observable due to their direct or indirect influence on the given low components.

A *labelling* $\mathcal{L}$ assigns to each atomic component (component without subcomponents) a label, high or low. [1] It is *sound* if it does not mark any component as high that can influence, and hence pass information to, a component marked as low. In the refinement order the labelling $\mathcal{L}'$ refines $\mathcal{L}$ ($\mathcal{L} \sqsubseteq \mathcal{L}'$), if low components in $\mathcal{L}$ are low also in $\mathcal{L}'$. The labelling $\mathcal{L}$ is *accurate*, if $\mathcal{L}$ is minimal in the refinement order such that $\mathcal{L}$ is sound and refines the initial labelling.

Determining whether a labelling is accurate is generally undecidable. Suppose $\mathbb{C}(P(x), s.C, 0)$ is assigned to a low component. Deciding whether C needs to be deemed low requires deciding whether there is some valid instantiation of x, such that $P(x)$ holds, which might not be decidable. However, it appears that in many cases, including those considered here, accurate labellings are feasible. In our approach we check the necessity of a label refinement by identifying an actual flow from the witness component to some low component. We cannot guarantee that this check always succeeds, for undecidability reasons. If it does not, the tool

[1]We have not found a use for ISA security lattices of finer granularity.

still tries to refine the low equivalence and a warning that the final relation may no longer be accurate is generated. For the considered case studies the tool always finds an accurate labelling, which is then by construction unique.

Labellings correspond to low-equivalence relations on pairs of states, relations that agree on all low components including the memory relation $\mathcal{R}_m$ and leave all other components unrestricted. *Noninterference* holds if the only components affecting the state or any return value are themselves low. Formally, assume the two pre-states s_1 and s_2 agree on the low-labelled components, expressed by a low-equivalence relation $\mathcal{R}$ on those states. Then, for a given transition Φ and preconditions $\mathcal{P}$, noninterference $\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi)$ holds if after Φ the post-states are again in $\mathcal{R}$ and the resulting return values are equal:

$$\begin{aligned}
&\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi) := \forall s_1, s_2, v_1, v_2, t_1, t_2 : \\
&\quad ((v_1, t_1) = \Phi s_1) \wedge ((v_2, t_2) = \Phi s_2) \wedge \mathcal{R}(s_1, s_2) \wedge \mathcal{P} s_1 \wedge \mathcal{P} s_2 \\
&\quad \Rightarrow \mathcal{R}(t_1, t_2) \wedge (v_1 = v_2)
\end{aligned}$$

Preconditions on the starting states can include architecture properties (version number, present extensions, etc.), a secure memory configuration and a specification of the privilege level. In our framework the user defines relevant preconditions and an initial low-equivalence relation $\mathcal{R}_0$ for an input ISA. The goal of the analysis is to statically and automatically find an accurate refinement of $\mathcal{R}_0$ so that noninterference holds for $\Phi = \texttt{NEXT}$. The analysis yields the final low-equivalence relation, the corresponding HOL4 noninterference theorem demonstrating the soundness of the relation, and a notification of whether the analysis succeeded to establish a guarantee on the relation's accuracy. The proof search is not guaranteed to terminate successfully, but we have found it robust enough to reliably produce accurate output on ISA models of considerable complexity (see Section C.5). We do not treat timing and probabilistic channels and leave safety-properties about unmodified components for future work.

C.3.2 Challenges

Our goal is to perform the analysis from an initial, user-supplied labelling on a standard ISA with minimal user interaction. In particular, we wish to avoid user supplied label annotations and error-prone manual rewrites of the ISA specification, that a type-based approach might depend on to eliminate some of the complications specific to ISA models. Instead, we address those challenges with symbolic evaluation and the application of simplification theorems. Since both are available in HOL4, and so are the models, we verify noninterference in HOL4 directly. This also frees us from external preprocessing and soundness proofs, thus minimizing the TCB. Below, we give examples for common challenges.

Representation The functional models that we use represent register sets as mappings. Static type systems for (purely) functional languages [89, 137] need to assign secrecy levels uniformly to all image values, even if a mapping has both public and secret entries. Adaptations of representation and type system might allow to type more accurately for lookups on constant locations. But common lookup patterns on locations represented by variables or complex terms would require a preprocessing that propagates constraints throughout large expressions.

Semantics Unprivileged ARMv7 processes can access the current state of the control register CPSR. The ISA specifies to (i) map all subcomponents of the control register to a 32-bit word and (ii) apply the resulting word to a bitmask. As a result, the returned value does actually not depend on all subcomponents of the CPSR, even though all of them were referred to in the first step. For accuracy, an actual understanding of the arithmetics is required.

reg'PSR s.CPSR	N	Z	C	V	Q	IT 1-0	J	re-served	GE 3-0	IT 7-2	E	A	I	F	T	M 4-0
&& 0xf8ff03df	1	1	1	1	1	0 0	0	1 1 1 1	1 1 1 1	0 0 0 0 0 0	1	1	1	1	0	1 1 1 1 1
=	N	Z	C	V	Q	0	0	re-served	GE 3-0	0	E	A	I	F	0	M 4-0

Figure C.1: Mapping an ARMv7 control register to a 32-bit word

Context-sensitivity Earlier work on ISA information flow [102] deals with ARM's complex operational semantics in a stepwise analysis, focusing on one subprocedure at a time. This allows for a systematic solution, but comes with the risk of insufficient context. For example, when reading from a register, usually two steps are involved: first, the concrete register identifier with respect to the current processor mode is looked up; second, the actual reading is performed. Analysing the reading operation in isolation is not accurate, since the lack of constraints on the register identifier would require to deem all registers low. In order to include restrictions from the context, [102] required a number of manual proofs. To avoid this, we analyse entire instructions at a time, using HOL4's machinery to propagate constraints.

C.4 Approach

We are not the first to study (semi-)automated hardware verification using theorem proving. As [51] points out for hardware refinement proofs, a large share of the proof obligations can be discharged by repeated unfolding (rewriting) of definitions, case splits and basic simplification. While easy to automate, these steps lead easily to an increase in complexity. The challenge, thus, is to find efficient and effective ways of rewriting and to minimize case splits throughout the proof. Our framework traverses the instruction set instruction by instruction, managing a task queue. For each instruction, three steps are performed: (i) rewriting/unfolding to obtain evaluated forms, (ii) attempting to prove noninterference for the instruction, (iii) on failure, using the identified counterexample to refine the low-equivalence relation. This section details those steps. After each refinement, the instructions verified so far are re-enqueued. The steps are repeated until the queue is empty and each instruction has successfully been verified with the most recent low-equivalence relation. Finally, noninterference is shown for `NEXT`, employing all instruction lemmas, as well as rewrite theorems for the fetch and decode transitions. Soundness is inherited from HOL4's machinery. Accuracy is tracked by the counterexample verification in step (iii).

C.4.1 Rewriting towards an Evaluated Form

The evaluated form of instructions is obtained through symbolic evaluation. Starting from the definition of a given transition, (i) let-expressions are eliminated, (ii) parameters of subtransitions are evaluated (in a call-by-value manner), (iii) the subtransitions are recursively unfolded by replacing them with their respective evaluated forms, (iv) the result is normalized, and (v) in a few cases substituted with an abstraction. Normalization and abstraction are described below. For the first three steps we reuse evaluation machinery from [68] and extend it, mainly to add support for automated subtransition identification and recursion. Preconditions, for example on the privilege level, allow to reduce rewriting time and the size of the result. Since they can become invalid during instruction execution, they have to be re-evaluated for each recursive invocation. Throughout the whole rewriting process, various simplifications are applied, for example on nested conditional expressions, case distinctions, words, and pairs, as well as conditional lifting, which we motivate below. For soundness, all steps produce equivalence theorems.

C.4.1.1 Step Library

The ISA models are provided together with so-called *step libraries*, specific to every architecture [68]. They include a database of pre-computed rewrite theorems, connecting transitions to their evaluated forms. Those theorems are computed in an automated manner, but are guided manually. Our tool is able to employ them as hints, as long as their preconditions are not too restrictive for the general security analysis. Otherwise, we compute the evaluated forms autonomously. Besides instruction specific theorems, we use some datatype specific theorems and general machinery from [68].

C.4.1.2 Conditional Lifting

Throughout the rewriting process, the evaluated forms of two sequential subtransitions might be composed by passing the result of the first transition into the formal parameters of the second. This often leads to terms like $\gamma(s) := \mathbb{C}(b, s[C_1 :=$

$c_1], s[C_2 := c_2]).C_3$. However, in order to derive equality properties in the noninterference proof (e.g., $[s_1.C_3 = s_2.C_3] \vdash \gamma(s_1) = \gamma(s_2)$) or to check validity of premises (e.g., $\gamma(s) = 0$), conditional lifting is applied:

$$\begin{aligned}
\gamma(s) &= \mathbb{C}(b, s[C_1 := c_1], s[C_2 := c_2]).C_3 && \text{lifting}\\
&= \mathbb{C}(b, (s[C_1 := c_1]).C_3, (s[C_2 := c_2]).C_3) && \text{simplifying}\\
&= \mathbb{C}(b, s.C_3, s.C_3) && \text{merging}\\
&= s.C_3
\end{aligned}$$

To mitigate exponential blow-up, conditional lifting should only be applied where needed. For record field accesses we do this in a top-down manner, ignoring fields outside the current focus. For example, in $\gamma(s)$ there is no need to process c_1 at all, even in cases where c_1 itself is a conditional expression.

C.4.1.3 Normalization

With record field accesses being so critical for performance, both rewriting and proof benefit from (intermediate) evaluated forms being normalized. A state term is *normalized* if it only consists of record field updates to a state variable s, that is, it has the form

$$s[C_1 := c_1, \ldots, C_n := c_n, R_1 := s.R_1[C_{1,1} := c_{1,1}, \ldots, C_{1,k} := c_{1,k}], \ldots].$$

For a state term τ updating state variable s in the fields $C_1, \ldots, C_n$ with the values $c_1, \ldots, c_n$, we verify the normalized form in a forward construction (omitting subcomponents here and below for readability; they are treated analogously):

$$\begin{aligned}
\tau &= \tau[C_1 := \tau.C_1, \ldots, C_n := \tau.C_n] && \text{(C.1)}\\
&= s[C_1 := \tau.C_1, \ldots, C_n := \tau.C_n] && \text{(C.2)}\\
&= s[C_1 := c_1, \ldots, C_n := c_n] && \text{(C.3)}
\end{aligned}$$

We significantly improve proof performance with the abstraction of complex expressions by showing (C.1) independently of the concrete τ and (C.2) independently of the values of the updates, both those inside τ and those applied to τ. We obtain $c_1, \ldots, c_n$ by similar means to those shown in the lifting example of γ above.

In [68], both conditional lifting and normalization are based on the precomputation of datatype specific lifting and unlifting lemmas for updates. Our procedures are largely independent of record types and update patterns. However, because of the performance benefits of [68], we plan to generalize/automate their normalization machinery or combine both approaches in future work.

C.4.1.4 Abstracted Transitions

Even with normalization, the specification of a transition grows quickly when unfolding complex subtransitions, especially for loops. We therefore choose to abstract selected subtransitions. To this end, we substitute their evaluated forms with terms that make potential flows explicit, but abstract away from concrete specifications. Let the normalized form of transition Φ be $\vec{\phi}s = (\beta(s), s[C_1 := \gamma_1(s), \ldots, C_n := \gamma_n(s)])$. The values of all primitive state updates $\gamma_1(s), \ldots, \gamma_n(s)$ on s and the return value $\beta(s)$ of Φ are substituted with new function constants $f_0, f_1, \ldots, f_n$ applied to relevant state components actually accessed instead of to the entire state:

$$\begin{aligned}\Phi s = \vec{\phi}s = \quad & (f_0(s.C_{0,1}, \ldots, s.C_{0,k_0}), \\ & s[C_1 := f_1(s.C_{1,1}, \ldots, s.C_{1,k_1}), \ldots, C_n := f_n(s.C_{n,1}, \ldots, s.C_{n,k_n})])\end{aligned}$$

Except for situations that suggest the need for a refinement of the low-equivalence relation, $f_0, \ldots, f_n$ do not need to be unfolded in the further processing of Φ. Low-equivalence of the post-states can be inferred trivially:

$$[(s_1.C_{1,1} = s_2.C_{1,1}) \wedge \ldots] \vdash f_1(s_1.C_{1,1}, s_1.C_{1,2}, \ldots) = f_1(s_2.C_{1,1}, s_2.C_{1,2}, \ldots))$$

To avoid accuracy losses in cases where $\vec{\phi}$ mentions components that neither return value nor low components actually depend on, we unfold abstractions as last resort before declaring a noninterference proof as failed.

C.4.2 Backward Proof Strategy

Having computed the evaluated form for an instruction Φ, we proceed with the verification attempt of $\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi)$ through a backward proof, for the user-provided preconditions $\mathcal{P}$ and the current low-equivalence relation $\mathcal{R}$. The sound backward proof employs a combination of the following steps:

- **Conditional Lifting:** Especially in order to resolve record field accesses on complex state expressions, we apply conditional lifting in various scopes (record accesses, operators, operands) and degrees of aggressiveness.

- **Equality of Subexpressions:** Let F be a functional component and n and m be two variables ranging over $\{0, 1, 2\}$. The equality

$$\begin{aligned} & \mathbb{C}(n = 2, 0, s_1.F(\mathbb{C}(n, a, b, c))) + s_1.F(\mathbb{C}(m, a, b, a)) \\ = \quad & \mathbb{C}(n = 2, 0, s_2.F(\mathbb{C}(n, a, b, c))) + s_2.F(\mathbb{C}(m, a, b, a))\end{aligned}$$

  can be established from the premises $s_1.F(a) = s_2.F(a)$ and $s_1.F(b) = s_2.F(b)$ by lifting the distinctions on n and m outwards or - alternatively - by case splitting on n and m. Either way, equality should be established for each summand separately, in order to limit the number of considered cases to $3 + 3$ instead of 3×3. Doing so in explicit subgoals also helps in discarding unreachable cases, such as the one where c would be chosen. We identify relevant expressions via pre-defined and user-defined patterns.

- **Memory Reasoning:** Axioms and derived theorems on noninterference properties of the memory subsystem and maintained invariants are applied.

- **Simplifications:** Throughout the whole proof process, various simplifications take effect, for example on record field updates.
- **Case Splitting:** Usually the mentioned steps are sufficient. For a few harder instructions or if the low-equivalence relation requires refinement, we apply case splits, following the branching structure closely.
- **Evaluation:** After the case splitting, a number of more aggressive simplifications, evaluations, and automatic proof tactics are used to unfold remaining constants and to reason about words, bit operations, unusual forms of record accesses, and other corner cases.

C.4.3 Relation Refinement

Throughout the analysis, refinement of the low-equivalence relation is required whenever noninterference does not hold for the instruction currently considered. Counterexamples to noninterference enable the identification of new components to be downgraded to low. When managed carefully, failed backward proofs of noninterference allow to extract such counterexamples. However, backward proofs are not complete. Unsatisfiable subgoals might be introduced despite the goal being verifiable. For accuracy, we thus verify the necessity of downgrading a component C before the actual refinement of the relation. To that end, it is sufficient to identify two witness states that fulfil the preconditions $\mathcal{P}$, agree on all components except C, and lead to a violation of noninterference in respect to the analysed instruction Φ and the current (yet to be refined) relation $\mathcal{R}$. We refer to the existence of such witnesses as $\overline{\mathcal{N}}$:

$$\begin{aligned}&\overline{\mathcal{N}}(\mathcal{R},\mathcal{P},\Phi,C) := \exists s, x_1, x_2, v_1, v_2, t_1, t_2 :\\ &\quad ((v_1,t_1) = \Phi(s[C := x_1])) \wedge ((v_2,t_2) = \Phi(s[C := x_2]))\\ &\quad \wedge \mathcal{P}(s[C := x_1]) \wedge \mathcal{P}(s[C := x_2]) \wedge (\neg\mathcal{R}(t_1,t_2) \vee (v_1 \neq v_2))\end{aligned}$$

If such witnesses exist, any sound relation $\mathcal{R}'$ refining $\mathcal{R}$ will have to contain some restriction on C. With the chosen granularity, that translates to $\forall s_1, s_2 : \mathcal{R}'(s_1,s_2) \Rightarrow (\mathcal{R}(s_1,s_2) \wedge s_1.C = s_2.C)$. We proceed with the weakest such relation, i.e., $\mathcal{R}'(s_1,s_2) := (\mathcal{R}(s_1,s_2) \wedge s_1.C = s_2.C)$. As discussed in Section C.3.1, it can be undecidable whether the current relation needs refinement. However, for the models that we analyzed, our framework was always able to verify the existence of suitable witnesses. The identification and verification of new low components consists of three steps:

1. **Identification of a new low component.** We transform subgoal G on top of the goal stack into a subgoal `false` with premises extended by $\neg G$. In this updated list of premises for the pre-states s_1 and s_2, we identify a premise on s_1 which would solve the transformed subgoal by contradiction when assumed for s_2 as well. Intuitively, we suspect that noninterference is prevented by the

disagreement on components in the identified premise. We arbitrarily pick one such component as candidate for downgrading.

2. **Existential verification of the scenario.** To ensure that the extended premises alone are not already in contradiction, we prove the existence of a scenario in which all of them hold. We furthermore introduce the additional premise that the two pre-states disagree on the chosen candidate, but agree on all other components. An instantiation satisfying this existential statement is a promising suspect for the set of witnesses for $\overline{\mathcal{N}}$. The existential proof in HOL4 refines existentially quantified variables with patterns, e.g., symbolic states for state variables, bit vectors for words, and mappings with abstract updates for function variables (allowing to reduce $\exists f : P(f(n))$ to $\exists x : P(x)$). If possible, existential goals are split. Further simplifications include HOL4 tactics particular to existential reasoning, the application of type-specific existential inequality theorems, and simplifications on word and bit operations. If after those steps and automatic reasoning existential subgoals remain, the tool attempts to finish the proof with different combinations of standard values for the remaining existentially quantified variables.

3. **Witness verification.** We use the anonymous witnesses of the existential statement in the previous step as witnesses for $\overline{\mathcal{N}}$. After initialisation, the core parts of the proof strategy from the failed noninterference proof are repeated until the violation of noninterference has been demonstrated.

In order to keep the analysis focused, it is important to handle case splits before entering the refinement stage. At the same time, persistent case splits can be expensive on a non-provable goal. Therefore, we implemented a depth first proof tactical, which introduces hardly any performance overhead on successful proofs, but fails early in cases where the proof strategy does not succeed. Furthermore, whenever case splits become necessary in the proof attempt, the framework strives to diverge early, prioritizing case splits on state components.

C.5 Evaluation

We applied our framework to analyse information flows on ARMv7-A and MIPS-III (64-bit RS4000). For ARM, we focus on user mode execution without security or virtualization extension. Since unprivileged ARM code is able to switch between several instructions sets (ARM, Thumb, Thumb2, ThumbEE), the information flow analysis has to be performed for all of them. For MIPS, we consider all three privilege modes (user, kernel, and supervisor). The single-core model does not include floating point operations or memory management instructions.

Table C.1 shows the initial and accurate final low-equivalence relations for the two ISAs with different configurations. All relations refine the memory relation. The *final relation* column only lists components not already restricted by the corresponding initial relations. For simplicity, the initial relation for MIPS restricts

ISA	mode	initial relation	final relation
ARMv7-A	user mode	program counter	user registers; control register `CPSR` (all flags); floating point registers of `FP.REG` and `FP.FSPCR`; `TEEHBR` register (coprocessor 14); `Encoding` ghost component; system control register `SCTLR` (coprocessor 15, flags: `EE`, `TE`, `V`, `A`, `U`, `DZ`)
MIPS-III	user *or* kernel *or* supervisor mode	program counter; `BranchTo`; `BranchDelay`; `CP0.Count`; exception marker; `CP0.Status.KSU`; `CP0.Status.EXL`; `CP0.Status.ERL`	all modelled system components
MIPS-III	*restricted* user mode		general purpose register set; `LLbit`; `lo`; `hi`; `CP0.Config.BE`; `CP0.Status.RE`; `CP0.Status.BEV`; `exceptionSignalled`

Table C.1: Identified flows (model components might deviate from physical systems)

ISA	rewrite	instr.	`NEXT`	total	
ARMv7	29,829	46,146	2,171	78,146	(21 h, 42 min)
MIPS (1)	537	1,790	1,594	3,921	(1 h, 5 min)
MIPS (2)	537	1,216	562	2,315	(38 min)

Table C.2: Proof performance (in seconds)

step	min	median	mean	max
rewrite	1	25	167	2,384
instr. (success)	1	15	96	3,605
instr. (fail)	3	26	72	1,544
refinement	7	50	89	1,326

Table C.3: Performance ARMv7 proof

three components accessed on the highest level of `NEXT`. The corresponding table cell also lists components already restricted by the preconditions. Initially unaware of the privilege management in MIPS, we were surprised that our tool first yielded the same results for all MIPS processor modes and that even user processes can read the entire state of system coprocessor `CP0`, which is responsible for privileged operations such as the management of interrupts, exceptions, or contexts. To restrict user privileges, the `CU0` status flag must be cleared (see last line of the table). While ARMv7-processes in user mode can not read from banked registers of privileged modes, they can infer the state of various control registers. Alignment control register flags (`CP15.SCTLR.A/U` in ARMv7) are a good example for implicit flows in CPUs. Depending on their values, an unaligned address will either be accessed as is, forcibly aligned, or cause an alignment fault.

Table C.2 shows the time that rewriting, instruction proofs (including relation refinement), and the composing proof for `NEXT` took on a single Xeon® X3470 core. The first benchmark for MIPS refers to unrestricted user mode (with similar times as for kernel and supervisor mode), the second one to restricted user mode. Even though we borrowed a few data type theorems and some basic machinery from the step library, we did not use instruction specific theorems for the MIPS verification. Both ISAs have around 130 modelled instructions, but with 9238 lines of L3 compared to 2080 lines [68], the specifications of the ARMv7 instructions are both larger and more complex. Consequently, we observed a remarkable difference in performance. However, as Table C.3 shows, minimum, median, and mean processing times (given in seconds) for the ARM instructions are actually moderate throughout all steps (rewriting, successful and failed noninterference proofs, and relation refinement). Merely a few complex outliers are responsible for the high verification time of the ARM ISA. While we believe that optimizations and parallelization could significantly improve performance, those outliers still demonstrate the limits of analyzing entire instructions as a whole. Combining our approach with compositional solutions such as [102] could overcome this remaining challenge. We leave this for future work.

C.6 Related Work

While most work on processor verification focuses on functional correctness [36, 51, 167] and ignores information flow, we survey hardware noninterference, both for special separation hardware and for general purpose hardware.

Noninterference Verification for Separation Hardware Wilding et al. [180] verify noninterference for the partitioning system of the AAMP7G microprocessor. The processor can be seen as a separation kernel in hardware, but lacks for example user-visible registers. Security is first shown for an abstract model, which is later refined to a more concrete model of the system, comprising about 3000 lines of ACL2. The proof appears to be performed semi-automatically.

SAFE is a computer system with hardware operating on tagged data [22]. Noninterference is first proven for a more abstract machine model and then transferred to the concrete machine by refinement. The proof in Coq does not seem to involve much automation.

Sinha et al. [163] verify confidentiality of x86 programs that use Intel's Software Guard Extensions (SGX) in order to execute critical code inside an SGX enclave, a hardware-isolated execution environment. They formalize the extended ISA axiomatically and model execution as interleaving between enclave and environment actions. A type system then checks that the enclave does not contain insecure code that leaks sensitive data to non-enclave memory. At the same time, accompanying theorems guarantee some protection from the environment, in particular that an adversary can not influence the enclave by any instruction other than a write to input memory. However, [163] assumes that SGX management data structures are not shared and that there are no register contents that survive an enclave exit and are readable by the environment. Once L3/HOL4 models of x86 with SGX are available, our machinery would allow to validate those assumptions in an automated manner, even for a realistic x86 ISA model. Such a verification would demonstrate that instructions executed by the environment do not leak enclave data from shared resources (like non-mediated registers) to components observable by the adversary.

Noninterference Verification for General Purpose Hardware Information flow analysis below ISA level is discussed in [138] and [112]. Procter et al. [138] present a functional hardware description language suitable for formal verification, while the language in [112] can be typed with information flow labels to allow for static verification of noninterference. Described hardware can be compiled into VHDL and Verilog, respectively. Both papers demonstrate how their approaches can be used to verify information flow properties of hardware executing both trusted and untrusted code. We are not aware of the application of either approach to information flow analysis of complex commodity processors such as ARM.

Tiwari et al. [173] augment gate level designs with information flow labels, allowing simulators to statically verify information flow policies. Signals from outside the TCB are modelled as *unknown*. Logical gates are automatically replaced with label propagating gates that operate on both known and unknown values. The authors employ the machinery to verify the security of a combination of a processor, I/O, and a microkernel with a small TCB. It is unclear to us how the approach would scale to commodity processors with a more complex TCB. From our own experience on ISA-level, the bottleneck is mainly constituted by the preprocessing to obtain the model's evaluated form and by the identification of a suitable labelling. The actual verification is comparatively fast.

In earlier work [102] we described a HOL4 proof for the noninterference (and other isolation properties) of a monadic ARMv7-model. A compositional approach based on proof rules was used to support a semi-automatic analysis. However, due to insufficient context, a number of transitions had to be verified manually or with the support of context-enhancing proof rules. In the present work, we overcome this issue by analysing entire instructions. Furthermore, our new analysis exhibits the low-equivalence relation automatically, while [102] provides it as fixed input. Finally, the framework described in the present paper is less dependent of the analysed architecture.

Verification of Binaries Fox's ARM model is also used to automatically verify security properties of binary code. Balliu et al. [23] does this for noninterference, Tan et al. [170] for safety-properties. Despite the seeming similarities, ISA analysis and binary code analysis differ in many respects. While binary verification considers concrete assembly instructions for (partly) known parameters, ISA analysis has to consider all possible assembly instructions for all possible parameters. On the other hand, it is sufficient for an ISA analysis to do this for each instruction in isolation, while binary verification usually reasons on a sequence (or a tree of) instructions. In effect, that makes the verification of a binary program an analysis on imperative code. In contrast, ISA analysis (in our setting) is really concerned with functional code, namely the operational semantics that describe the different steps of single instructions. In either case, to enable full automation, both analyses have to include a broader context when the local context is not sufficient to verify the desired property for a single step in isolation. As discussed above, we choose an instruction-wide context from the beginning. Both [23] and [170] employ a more local reasoning. In [170] a Hoare-style logic is used and context is provided by selective synchronisation of pre- and postconditions between neighbouring code blocks. In [23] a forward symbolic analysis carries the context in a path condition when advancing from instruction to instruction. SMT solvers then allow to discard symbolic states with non-satisfiable paths.

C.7 Discussion on Unpredictable Behaviour

ISA specifications usually target actors responsible for code production, like programmers or compiler developers. Consequently, they are often based on the assumption that executed code will be composed from a set of well-defined instructions and sound conditions, so that no one relies on combinations of instructions, parameters and configurations not fully covered by the specification. This allows to keep instructions partly underspecified and leave room for optimizations on the manufacturer's side. However, this practice comes at the cost of actors who have to trust the execution of unknown and potentially malicious third-party code. For example, an OS has an interest in maintaining confidentiality between processes. To that end, it has different means such as clearing visible registers on context switches. But if the specification is incomplete on which registers actually are visible to an instruction with uncommon parameters, then there is no guarantee that malicious code can not use underspecified instructions (i.e., instructions resulting in unpredictable states) to learn about otherwise secret components. ARM attempts to address this by specifying that "*unpredictable* behaviour must not perform any function that cannot be performed at the current or lower level of privilege using instructions that are not *unpredictable*" [13]. While this might indeed remedy integrity concerns, it is still problematic for noninterference. An underspecified instruction can be implemented by two different "safe" behaviours, with the choice of the behaviour depending on an otherwise secret component. The models by Fox et al. mark the post-states of underspecified operations as unpredictable by assigning an exception marker to those states. In addition, newer versions still model a reasonable behaviour for such cases, but there is no guarantee that the manufacturer chooses the same behaviour. A physical implementation might include flows from more components than the model does, or vice versa. A more conservative analysis like ours takes state changes after model exceptions into account, but can still miss flows simply not specified. To the rescue might come statements from processor designers like ARM that "*unpredictable* behaviour must not represent security holes" [12]. In one interpretation, flows not occurring elsewhere can be excluded in underspecified instructions. The need to rely on this interpretation can be reduced (but not entirely removed) when the exception marker itself is considered low in the initial labelling. As an example, consider an instruction that is well-defined when system component C_1 is 0, but underspecified when it is 1. The manufacturer might choose different behaviours for both cases, thus possibly introducing a flow from C_1 to low components. At the same time, the creator of the formal model might implement both cases in the same way, so that the analysis could miss the flow. But with a low exception marker, C_1 would also be labelled low, since it influences the marker. However, an additional undocumented dependency on another component C_2 that only exists when C_1 is 1 can still be missed.

To obtain security guarantees on their products, manufacturers should either follow an already analyzed model when implementing underspecified behaviour or adapt/create models according to their implementation choices and analyze those new models with the machinery presented in this work or with similar tools.

C.8 Conclusions and Future Work

We presented a sound and accurate approach to automatically and statically verify noninterference on instruction set architectures, including the automatic identification of a least restrictive low-equivalence relation. Besides applying our framework to more models such as the one of ARMv8, we intend to improve robustness and performance, and to cover integrity properties as well.

Integrity Properties We plan to enhance the framework by safety-properties such as nonexfiltration [90, 102] and mode switch properties [102]. Nonexfiltration asserts that certain components do not change throughout (unprivileged) execution. Mode switch properties make guarantees on how components change when transiting to higher privilege levels, for example that the program counter will point to a well-defined entry point of the kernel code. We believe that both properties can be derived relatively easily from the normalized forms of the instructions.

Performance Optimization While our benchmarks have demonstrated that ISA information flow analysis on an instruction by instruction basis allows for a large degree of automation, they also have shown that this approach introduces severe performance penalties for more complex instructions. To increase scalability and at the same time maintain automation, we plan to investigate how to combine the compositional approach of [102] with the more global reasoning demonstrated here. Furthermore, there is potential for improvements in the performance of individual steps. E.g., our normalization could be combined with the one of [68].

Paper D

Formal Verification of Secure User Mode Device Execution with DMA

Oliver Schwarz and Mads Dam

Abstract

Separation between processes on top of an operating system or between guests in a virtualized environment is essential for establishing security on modern platforms. A key requirement of the underlying hardware is the ability to support multiple partitions executing on the shared hardware without undue interference. For modern processor architectures - with hardware support for memory management, several modes of operation and I/O interfaces - this is a delicate issue requiring deep analysis at both instruction set and processor implementation level. In a first attempt to rigorously answer this type of questions we introduced in previous work an information flow analysis of user program execution on an ARMv7 platform with hardware supported memory protection, but without I/O. The analysis was performed as a semi-automatic proof search procedure on top of an ARMv7 ISA model implemented in the Cambridge HOL4 theorem prover by Fox et al. The restricted platform functionality, however, makes the analysis of limited practical value. In this paper we add support for devices, including DMA, to the analysis. To this end, we propose an approach to device modeling based on the idea of executing devices nondeterministically in parallel with the (single-core) deterministic processor, covering a fine granularity of interactions between the model components. Based on this model and taking the ARMv7 ISA as an example, we provide HOL4 proofs of several noninterference-oriented isolation properties for a partition executing in the presence of devices which potentially use DMA or interrupts.

D.1 Introduction

Modern computing platforms usually execute multiple kinds of services together. Entertainment software runs next to online-banking applications. Personal communication services run next to business software. For security, there is a strong need to execute processes in isolation from each other, such that mutual influence is minimized and their integrity and confidentiality fully protected. Some approaches attempt to achieve this level of isolation within the commodity operating system, while others base upon separation kernels, micro kernels or virtualization. In all cases, the hardware platform is required to allow for strong compartmentalization of process execution. Untrusted processes should neither be able to influence processes at higher trust levels nor to learn anything about their state of execution. Basic protection is enabled by several privilege rings of operation and memory protection/management units (MPU/MMU), controlled by control registers, coprocessors and configurations in memory. Information can potentially flow via multiple system components and operations, such as memory accesses by the CPU, directly accessible registers, side effects of control registers, coprocessors, timing channels, device ports, device accesses to memory, or interrupts, to just name a few. Therefore it is crucial to understand and document the information flows that are possible on a complex platform. These flows are not always obvious. For example, on some x86 processors it is possible for low-privilege code to overwrite higher privilege code by writing to an address that usually refers to the video card [62]. To enable this attack, it suffices to flip a configuration bit usually accessible from the low privilege level. On ARM, comparison instructions change flags in the current program status register (CPSR). When switching processes, those flags therefore need to be cleared or reloaded from the register bank of the invoked process. Peripheral devices further increase a system's complexity. Assigning them to only one process per device is sometimes insufficient to prevent information flow between processes. If a device has the capability of performing direct memory access (DMA), it can be programmed to circumvent the access policy of the MMU unless advanced hardware support for virtualization is provided and this support is soundly configured, which is by no means self-evident. Even if the configuration of DMA controllers is monitored to prevent copying between partitions, undesired information flows can still occur. For example, a device can fire an interrupt depending on the content of memory controlled by a user process, allowing for side channel communication based on the delays introduced by such interrupts. Given the complexity of modern hardware, it is not trivial to avoid misconfiguration. In previous work [102] we introduced a formal information flow analysis of ARMv7 user mode execution on instruction set architecture (ISA) level, however, not yet covering devices. With devices, the system's state increases and so does the set of possible information flows. CPU and multiple DMA devices with unknown behaviour can execute in parallel, possibly accessing the same memory, with an unknown interleaving.

This paper presents the following contributions. First, we extend the Cambridge HOL4 model of the ARM architecture [70] by a general device framework. To the best of our knowledge, this is the first theorem prover model for devices capable of reasoning on DMA. It is sufficiently detailed to capture possible information flows on modern systems. The adaptation to other processor architectures can be done with a minor effort. Second, we identify several secure device configurations. Since the focus is on platform information flow security rather than functionality, we do not restrict verification to concrete device specifications, but provide a suitable abstraction. For the verification of a system's separation properties, it is then sufficient to show that the configuration of the system devices complies with the identified abstract requirements. Finally, based on the proposed configurations and the device framework, we prove the following partitioning-related properties of the ARMv7 architecture with devices:

1. *Non-infiltration* states that the user mode execution of an ARMv7 processor is independent of devices that neither write to the memory accessed by the active process nor fire interrupts.

2. The integrity property of *extended non-exfiltration* states in turn that user mode processes are unable to influence devices that do not read from CPU-modifiable memory. Moreover, other protected resources[1], such as memory of neighboring processes, can not be modified by the process. That is true even if dedicated peripheral devices do access these resources in parallel. More specifically, the transformation of these resources depends only on such dedicated and inaccessible devices.

3. The third property, *filtered device non-infiltration*, states that devices which operate on disjunct resources can not influence each other without the interaction of the CPU.

One of the added challenges in the formulation and verification of the properties compared to [102] is that - with CPU and devices operating in parallel - different principles can cause different effects on the shared state. Covering separation during user mode execution, the results can be applied in the verification of hypervisors, separation kernels and operating systems. To the best of our knowledge, this is the first work on non-interference like platform properties for autonomous device execution.

D.2 Related Work

Hillebrand et al. [92] describe a pen and paper model, later formalized in Isabelle/HOL [4], for a memory-mapped hard disk integrated with a RISC architecture. The model includes side effects on device port reads/writes, interrupts and an external environment. The latter is also used to realize non-determinism, especially in timing matters. Direct memory access is not covered. Furthermore, unlike ARM,

[1]See Section D.6.3 for the complete list of protected resources.

the processor model does not perform multiple memory operations per instruction (instruction fetches are assumed to not refer to device ports), which allows for executing processor and device steps in an interleaved way after one another, without considering device progress within a single processor step. In [4] they describe the exploitation of an oracle that enables the sequentialization of the concurrent execution of devices and CPU. While the concrete order of events in a system is hard to predict, this oracle allows for the quantification over all execution orders and external inputs. These results were applied in the functional correctness verification of a microkernel [6]. The system architecture includes concurrent devices; besides a hard disk used for page fault handling also devices accessible by user processes are considered. Using refinement techniques, the authors were able to establish a simulation relation between an abstract microkernel programming framework and the instruction level. On the abstract levels devices are represented as ghost data structures.

Duan and Regehr [61] describe a general device model framework integrated with the HOL4 model for ARM6 by Anthony Fox [67] in a lock-step manner. They provide a proof of concept for a UART device and its driver, presenting statements on functionality, (memory) safety and timing. Similar to [92] and [4], they model side effects of memory mapped accesses to device ports and exploit input streams. Again, DMA is not supported. The authors prove that the integration of new devices to the system does not cause new undefined behaviour and preserves already established system predicates. This allows to verify driver correctness for one device at a time, but clearly would not hold for DMA devices. In his PhD thesis [60], Duan integrates his model into the Cambridge model of ARMv7 and adds reasoning about interrupts. Since ARMv7 has instructions that perform multiple memory accesses, device port reads/writes have been integrated into the primitives for memory accesses. Also autonomous device transitions are integrated into the execution cycle, however, they occur only once per instruction. In a DMA setting this is not sufficient since physical memory can be changed by devices between two memory accesses from the CPU side. In order to reason about DMA with a finer granularity and to allow for non-deterministic device progress, we propose a different model in Section D.5.

Monniaux modelled a USB controller in C and used an extended version of the Astrée static analyzer to verify that neither controller nor its driver will transfer data incorrectly [122]. He includes asynchronous DMA into his reasoning. By modelling the controller's behaviour with non-deterministic choices, an over-approximation is achieved. Isolation from untrusted software is not discussed.

XMHF [175] is a hypervisor framework for x86 exploiting virtualization support, in particular the DMA protection of Intel Vt-d and AMD's device exclusion vectors. The framework allows unmodified guests direct device access. System devices are included in the attacker model. Exploiting the model checker CBMC, mainly memory integrity is verified. As for direct memory access, CBMC verifies that the control register value written to the DMA protection hardware register has the bit set which enables DMA protection. The DMA table is manually audited. However,

$$\begin{array}{lll}
\texttt{arm\_state} = < | & \texttt{psrs} & : \texttt{PSRName} \to \texttt{ARMpsr}; \\
 & \texttt{regs} & : \texttt{RName} \to \texttt{word32}; \\
 & \texttt{memory} & : \texttt{word32} \to \texttt{word8}; \\
 & \texttt{coproc} & : \texttt{coprocessors}; \\
 & \texttt{accesses} & : \texttt{memory\_accesslist}; \\
 & \texttt{misc} & : \texttt{Monitors} \# \texttt{ARMinfo} \# \texttt{bool} \# \texttt{bool} \mid >;
\end{array}$$

Figure D.1: The ARM state in HOL4

it seems that the actual effects of devices or the DMA protection unit are not part of the model. In the present paper we focus on systems without hardware support for virtualization.

The properties shown in this paper are inspired by Heitmeyer et al. [90], who formulated non-infiltration and non-exfiltration for a separation kernel. We adapt those properties to a platform with DMA devices and a CPU in user mode.

D.3 The HOL4 ARM Model

We base our work on Fox et al's monadic HOL4 ISA model [70] of ARMv7 platforms without hardware extensions such as TrustZone or virtualization support. Figure D.1 shows a simplified definition of the processor state in this model. The function `psrs` returns the value of a processor state register (of type `ARMpsr`). The processor state registers include the current program status register, `CPSR`, in addition to the banked psrs `SPSR_m` for each privileged mode `m`, except for system mode. The ARMv7 core provides seven processor modes: one non-privileged user mode `usr`, and six privileged modes, activated when an exception (such as an interrupt) is invoked. The function `regs` takes a register name and returns its value. The ARM registers include sixteen general purpose registers ($\texttt{r0} - \texttt{r15}$) that are available from all modes in addition to the banked registers of each privileged mode that are available only in that mode. The function `memory` maps an address (`word32`) to a byte (`word8`). Caches are not represented in the model. The field `coproc` represents the set of coprocessor registers in `CP14` and `CP15` implicitly influencing execution, to a large extent even user-mode/exception execution. The field `misc` represents exclusive monitors for synchronization purposes, general information about the state, e.g. the architecture version, if the system is waiting for an interrupt etc, and `accesses` records the accesses to the memory.

A *computation* in the monadic HOL4 ARM model is a term of the type

$$\alpha\ \texttt{M} = \texttt{arm\_state} \mapsto (\alpha, \texttt{arm\_state})\ \texttt{error\_option}$$

where `error_option` is a datatype defined as:

$$(\alpha, \beta)\ \texttt{error\_option} \quad = \quad \texttt{ValueState of } \alpha \Rightarrow \beta \mid \texttt{Error of string}$$

Computations act on a state (`arm_state`) and return either `ValueState` a s, a new state s along with a return value a of type α, or an error e (if the computation is underspecified by the ARM specification). The monad unit `constT` injects a value into a computation, i.e. `constT` a s = `ValueState` a s, while binding is a sequential composition operation

$$\begin{aligned} f_1 \gg\!=_e f_2 \quad = \quad & \lambda s.\texttt{case}\ f_1\ s\ \texttt{of Error}\ c \to \texttt{Error}\ c \\ & \quad \mid \texttt{ValueState}\ a\ s' \to \texttt{if}\ e\ s'\ \texttt{then}\ f_2\ a\ s'\ \texttt{else}\ f_1\ s. \end{aligned}$$

That is, if e holds in the final state of f_1, the return value of f_1 is passed to f_2 as the input parameter, otherwise f_2 is not executed. In addition to unit and binding, the ARM monadic specification uses standard constructs for lambda, full conditional, `let`, and `case`, as well as the monad operations parallel composition, positive conditional (`condT` e f = `if` e `then` f `else constT` ()), error (`errorT` a = `Error` a), and an iterator. Values of state components can be obtained and set by `readT` $f = \lambda y.$(`ValueState` $(f\ y)\ y$) and `writeT` $f = \lambda y.$(`ValueState` $()\ (f\ y)$).

D.4 Memory Management

The Memory Management Unit (MMU) enforces memory access policies and is therefore crucial for isolation. MMU configurations consist of page tables in memory and dedicated registers of `CP15`. Specific to ARM is the possibility of partitioning pages into collections of memory regions (*domains*), each representing one security role. The coprocessor registers involved are `SCTLR`, `TTBR0` and `DACR`. The `SCTLR` register determines whether the MMU is enabled, `TTBR0` contains the base address of the page table, and `DACR` manages the ARM domains.

In [102] we extended the ARM model with MMU functionality. The extended model defines two key functions, `permitted`, to account for access permissions, and `mmu_setup`, to reflect a "good configuration" property. The permission evaluation function `permitted` a b_w $(v_s,\ v_t,\ v_d)$ b_p m takes as parameters an address a, a flag b_w indicating whether reading or writing access is to be evaluated, the values of `SCTLR`, `TTBR0` and `DACR`, a flag b_p indicating whether permissions are to be checked against a privileged mode, and the memory m containing the page tables. The pair of booleans returned by `permitted` states whether the access permission on the specified location is defined in the given configuration, and the outcome of that decision (`true` if access is granted). Here, we apply a basic version of `permitted`, supporting one-level page tables with an identity address translation, but including the interpretation of ARM domains. It is shown that `permitted` is defined for all addresses in all reachable states.

The history of memory accesses is tracked in the `accesses` ghost field of the machine state, allowing to compute the set of memory locations accessed by an instruction. To stop computation after the first access violation, $\gg\!=_{\texttt{nav}}$ has been chosen as standard binding operator. The property `nav` s holds if there is no access violation recorded in state s. Formally, this is the case if there is no entry in the access list of machine state s that causes `permitted` to return a negative answer in the current configuration of s. The recording of an access always happens before the access itself.

We finally need to formulate a suitable well-formedness condition for the MMU configuration. Let `accessible` i a express that address a is readable and writable by user process i. Other, more refined, static user level access policies can be supported with minor effort. The predicate `mmu_setup` i s holds if (i) the MMU configuration $((d, p) =$ `permitted` a b_w (`mmu_registers` s) $\perp$ s.`memory`) for any address a and access type b_w is defined (i.e., d is `true`), (ii) the state s implements the desired access policy for process i (i.e., $p =$ `accessible` i a), and (iii) none of the active page tables in s (represented by the address set `page_table_adds` s) is accessible according to the policy.

$$\begin{aligned}&\texttt{mmu\_setup}\ i\ s :=\\ &\forall a, w, d, p.\ ((d, p) = \texttt{permitted}\ a\ b_w\ (\texttt{mmu\_registers}\ s)\ \perp\ s.\texttt{memory})\\ &\quad \Rightarrow d \wedge (p = \texttt{accessible}\ i\ a)\ \wedge\ (a \in (\texttt{page\_table\_adds}\ s) \Rightarrow \neg\texttt{accessible}\ i\ a)\end{aligned}$$

For the properties shown in Section D.6 we furthermore prohibit user space processes to access device ports by assuming that the (state-independent) set of device addresses and `accessible` addresses are disjoint for every user process.

D.5 Device Model Framework

We present a general device model framework, capable of reasoning on DMA devices and with the ambition to cover all possible executions of a platform where the single-core processor and multiple devices run in parallel. In practice, changes to shared resources such as memory happen asynchronously and in a practically unpredictable order. We apply a non-deterministic approach that takes into account all possible interleavings and - to be conservative on timing behaviour - all possible durations of device and CPU actions, without restrictions on deadlines. Naturally, this does not allow to reason on whether an operation will be finished before a certain event or not. A timing accurate model would need to take CPU and system implementation specific details into account, including caches, MMU implementation specifics (such as the translation lookaside buffer), pipeline architecture and bus contention protocols. Models at this level of detail are surely interesting, but likely to be vastly more complex. The main challenge when integrating DMA into a device model is that memory can potentially change at any time, for example, between reading two words belonging to a multiple load instruction. Also inter-device communication can occur in any order and granularity. This precludes models that synchronize CPU and devices only between different CPU instructions. To address this challenge and allow for asynchronous device execution, we augment the CPU model with an abstract scheduler as suggested in [4], an oracle of the type

$$\texttt{oracle} : \texttt{num} \rightarrow (\texttt{dev\_name}\ \#\ \texttt{word32 option})\ \texttt{option}$$

The oracle provides a non-deterministic sequence of activity entries where the n-th activity entry `oracle` n is either `NONE` (then the CPU is progressed rather than a device) or a tuple `SOME` (d, e_{iopt}), indicating the device with identifier (`dev_name`)

d to progress one step, possibly in the context of the optional external 32-bit input e_{iopt}. We assume processor liveness: $\forall n.\ \exists m.\ (m \geq n) \wedge (\texttt{oracle}\ m = \texttt{NONE})$. Liveness of devices can be optionally included, but is not required for the properties we show in this paper. To include devices into the machine state, `arm_state` is extended by the following components:

```
devices    : dev_rec;
ext_out    : dev_name → word32 list;
int_fired  : bool;
counter    : num
```

The record `devices` subsumes the states of all devices [2]. The external output is represented by a finite stream of 32-bit words for each device, accessible via the map `ext_out`, mapping each device identifier to the list of outputs produced so far for that device. Whether an interrupt has been fired during the current execution cycle is stored in `int_fired`. Fast interrupts or advanced interrupt controllers are not part of the model. Finally, `counter` points to the current position in the oracle index and is incremented every time the oracle is invoked. Devices can progress in one of four ways:

- *Autonomously*: A device may make processor-independent progress, either by entirely internal actions or by receiving external inputs, accessing memory, raising interrupts, or producing external outputs. The function

  ```
  progress : device ↦ (word32 option)
    ↦ (mem_req option # bool # word32 option # device) option
  ```

  takes as arguments a device state D and a possible external input e_{iopt}. It returns either an "error" (`NONE`) representing undefined behaviour or a tuple $(r_{opt}, b_{int}, e_{oopt}, D')$ with an optional read/write access request r_{opt} to the system's memory bus (including an address and the access type), a flag b_{int} indicating a possible interrupt, an optional external output e_{oopt} and the updated device state D'. This function is used to progress devices with a non-deterministic frequency after every executed CPU instruction and between memory accesses made by the CPU or other devices.

- *Upon reception of a pending reply to a memory bus read*: As a result of an autonomous step, a device can send a read request to the bus, in order to read from the system memory or from the port of another device. The result is communicated to the device by invoking the `receive` operation:

  ```
  receive : device ↦ mem_req ↦ mem_answer ↦ device option
  ```

  For a given device state D and request r being answered, `receive` $D\ r\ v$ passes the read value v (as either byte or word) to D and returns either an error (`NONE`) or the updated device D'. Write operations requested by devices

[2]We notate `devices`.d for the state of the device with identifier d in the record `devices`.

do not have an answer and thus change only the memory, but not the device. We assume that reads are atomic operations and the memory bus will always complete an issued read before handling new operations. In other words, we exclude scenarios where a device notices that one of its ports is being read and already starts side effect computations affecting memory or other system components without first returning the requested value. That is no limitation for the properties we show in this paper, since we do not consider port accesses in them.

As for device reads from physical memory, we refer to the discussion in Section D.8.

- *As side effect on port reads*: The CPU or another device may read from an address that is mapped to a device. This address can belong to a device register, but in general it is not required that such a register is physically existing, for example when the address is associated with a side effect. We therefore use the general term *port* rather than register. We assume atomic 32-bit accesses to device ports and that port accesses are not cached. The function

  $$\texttt{d\_read} : \texttt{device} \mapsto \texttt{word32} \mapsto (\texttt{word32} \# \texttt{device})\ \texttt{option}$$

  takes as arguments a device state D and the port number indicating which port of the device is to be read. A special data structure of the model maps any virtual address to either physical memory or a device identifier together with a port number. The result of `d_read` is either `NONE` or the read 32-bit value together with a possibly updated device state D'.

- *As side effect on port writes*: the function

  $$\texttt{d\_write} : \texttt{device} \mapsto \texttt{word32} \mapsto \texttt{word32} \mapsto \texttt{device option}$$

  takes as arguments a device state D, the port number indicating which port of the device is to be written to and the 32-bit value to be written. It returns either an error (`NONE`) or the updated device state D'.

Different types of devices will have different behaviour. That is, the concrete functionalities of the device functions depend on the addressed device. While `d_write` and `d_read` are integrated into the existing memory access primitives of the ARM model (similar to [60]), `progress` and `receive` are used to realize autonomous progress of devices. Figure D.2 defines `advance_single` f n that uses the oracle at position n to determine the next device to progress autonomously and that updates the state with the effect of this progress accordingly. Subsequently, resulting memory requests are realized (including possible side effects when directed to other devices) and finally `counter` is increased. A filtering predicate f can be used to apply those steps only to devices d for which f d holds. Here, `update_device` and `update_output` update the `devices` and `ext_out` components of the current state,

```
advance_single $f$ $n$ := readT ($\lambda s.\ s$.devices) $\gg\!=_{\mathtt{T}}$
                          ($\lambda \hat{D}$. (case oracle $n$ of NONE $\Rightarrow$ constT ()
                           |SOME $(d, e_{iopt}) \Rightarrow$
                            condT ($f$ $d$)
                              (case progress $\hat{D}.d$ $e_{iopt}$ of NONE $\Rightarrow$ errorT $\epsilon$
                               |SOME $(r_{opt}, b_{int}, e_{oopt}, D') \Rightarrow$
                                 update_device $d$ $D'$ $\gg\!=_{\mathtt{T}}$
                                 ($\lambda u$. update_output $d$ $e_{oopt}$ $\gg\!=_{\mathtt{T}}$
                                 ($\lambda u$. condT $b_{int}$
                                   (writeT ($\lambda s.\ s$ with int_fired := T)) $\gg\!=_{\mathtt{T}}$
                                   ($\lambda u$. case $r_{opt}$ of NONE $\Rightarrow$ constT ()
                                     |SOME $r \Rightarrow$ mem_acc_by_dev $r$ $d$))))) $\gg\!=_{\mathtt{T}}$
                         ($\lambda u$. increment_counter))
```

Figure D.2: The `advance_single` computation.

respectively, and `mem_acc_by_dev` r d realizes the memory access request r on behalf of device d. Our model does not include any IOMMU. The repeated execution of `advance_single` is realized by `advance`, where for $n > 0$, `advance` f n traverses the oracle with filtering predicate f up to oracle position n and `advance` f 0 traverses the oracle until a `NONE` as activity entry indicates that execution will continue on the CPU side. The `advance` computation will synchronize devices and CPU before each memory bus access (for memory mapped ports and physical memory) of the CPU[3] and additionally between two execution cycles. The model supports instruction fetching from device addresses, but we assume that page table walks are not performed on device ports. In the properties shown in this paper we assume an MMU setting that prohibits both, by choosing device addresses, page table addresses and user space accessible memory to be disjoint.

Incorporating the MMU and device extension, the instruction execution function `next` (Fig. D.3) involves the following functionality: if an interrupt is pending and not masked, an interrupt exception is taken. Otherwise, the CPU may (if requested so by the previous instruction) wait for an interrupt or fetch and execute the next instruction pointed to by the program counter. If an access violation is recorded during instruction fetching or execution, a prefetch or data abort exception is initiated. The access list is cleared between the single steps and unconditional binding $\gg\!=_{\mathtt{T}}$ is used occasionally, preventing the execution from halting and instead allowing the initiation of exceptions and the detection of possible further violations. In addition to the synchronization phases before any of the CPU's memory operations, possible autonomous device steps are considered after each instruction execution,

[3]Consequently, accesses to the shared state, in particular the memory bus, determine the granularity of the system.

```
next := (clear_alist ≫=
        (λu. readT (λs. s.int_fired ∧ ¬s.psrs(0, CPSR).I) ≫=
        (λb. if (¬b) then
               waiting_for_interrupt ≫=
               (λw. condT (¬w)
                      (fetch_instruction ≫=_T
                      (λ(o, i). is_viol ≫=_T (λa. clear_alist ≫=
                      (λu. if a then prefetch_abort
                           else (execute i ≫=_T (λu. is_viol ≫=_T
                                (λa. condT a
                                     (clear_alist ≫=
                                     (λu. data_abort)))))))))) ≫=_T
               (λu. advance all 0)
            else take_irq_exception ≫= (λu. clear_interrupts))))
```

Figure D.3: The `next` computation.

in order to account for interrupt initiations.

As discussed earlier, our model is not clock accurate. While this is common with related work, usually a fixed duration is assumed for all instructions [60]. In our model, durations are non-deterministic, controlled by the oracle. However, given a specific oracle sequence, memory extensive instructions generally consume more oracle entries (i.e, time). For the properties of this paper and the targeted abstraction level, concrete instruction time is not relevant.

D.6 Security Properties

We next turn to formalizing several partitioning properties in terms of non-infiltration and non-exfiltration (cf. [90]), adapted to our setting, i.e., arbitrary and unknown user mode code executing on an ARMv7 CPU and in parallel with DMA devices. The isolation does not rely on an IOMMU. Together with a proper separation kernel (configuring devices, mediating user registers etc.) the discussed properties allow for establishing full process isolation within a system.

D.6.1 Suitable Device Configurations

Since isolation between CPU and DMA devices requires controlled device behaviour, we first describe possible device configurations that we consider secure. They allow the devices to change their internal state in an arbitrary way, but impose restrictions on DMA and interrupts. Kernels are responsible for realizing such a device configuration, in order to guarantee that process isolation is maintained when yielding to user mode. We expect those configurations to stay preserved throughout the whole user mode execution (while access to device ports is forbidden to both CPU and other devices). Formally, a configuration C is called *invariant* if it is preserved over autonomous steps, including the reception of replies to autonomously issued read requests:

$$\begin{aligned}&\texttt{invariant}\ C := \ \forall D.\ C\ D\ \Rightarrow\\ &\quad(\forall e_{iopt},\ r_{opt},\ b_{int},\ e_{oopt},\ D'.\\ &\quad\quad(\texttt{progress}\ D\ e_{iopt} = \texttt{SOME}\ (r_{opt}, b_{int}, e_{oopt}, D')) \Rightarrow C\ D')\\ &\quad\wedge\ (\forall r,\ v,\ D'.\ (\texttt{receive}\ D\ r\ v = \texttt{SOME}\ D') \Rightarrow C\ D')\end{aligned}$$

A property P holds on a device D in a *stable* way if it is established by an invariant configuration C:

$$\texttt{stable}\ P\ D := \exists C.\ \texttt{invariant}\ C\ \wedge\ C\ D \wedge\ (\forall D'.\ C\ D' \Rightarrow P\ D')$$

The stable properties we are interested in guarantee that devices are configured in a way that prevents them from communicating with other devices, running into an undefined state, accessing memory out of well-defined boundaries or firing interrupts in dependency on DMA operations. We believe that many devices (e.g., timers or DMA controllers) can be configured to respect those restrictions. The `restricted_dma` predicate holds if a device is configured to restrict its DMA requests to a set A of memory addresses.

$$\begin{aligned}&\texttt{restricted\_dma}\ A\ D := \forall e_{iopt},\ r,\ b_{int},\ e_{oopt},\ D'.\\ &\quad(\texttt{progress}\ D\ e_{iopt} = \texttt{SOME}\ (\texttt{SOME}\ r, b_{int}, e_{oopt}, D')) \Rightarrow (\texttt{access\_request\_map}\ r \subseteq A)\end{aligned}$$

Here, `access_request_map` maps a memory request to the set of byte addresses it involves. A device is called `silent` if A does not include device ports. Devices not firing interrupts are called `interrupt_free`. We say that a device is `errorfree`, if `progress` and `receive` do not return `NONE` for any inputs. Based on those properties, we distinguish three specific device configurations: devices involving DMA operations on the memory of the active process, devices involving DMA operations on the memory of other processes, and devices that are allowed to fire an interrupt.

$$\begin{aligned}&\texttt{own\_devices}\ i\ D := \texttt{stable}\ (\texttt{restricted\_dma}\ (\texttt{own\_add}\ i))\ D\\ &\qquad\qquad\qquad\quad \wedge\ \texttt{stable interrupt\_free}\ D\\ &\texttt{foreign\_devices}\ i\ D := \texttt{stable}\ (\texttt{restricted\_dma}\ (\texttt{foreign\_add}\ i))\ D\\ &\qquad\qquad\qquad\qquad\quad \wedge\ \texttt{stable interrupt\_free}\ D\\ &\texttt{interrupt\_devices}\ D := \texttt{stable}\ (\texttt{restricted\_dma empty\_set})\ D\end{aligned}$$

Here, `own_add` i is the set of addresses belonging to process or partition i, while `foreign_add` i spans exactly over the other user partitions. We do not allow a device to do both, accessing memory and firing interrupts. This is to prevent information flow from a user process' memory to another process' perception of execution time. [4] For a given user process i we assign each device d to one of

[4]Alternative configurations could allow DMA devices to fire interrupts, as long as those interrupts are masked while foreign processes are executing. However, this requires a very careful and more complex design at kernel level to avoid timing channels when interrupts occur close to context switches.

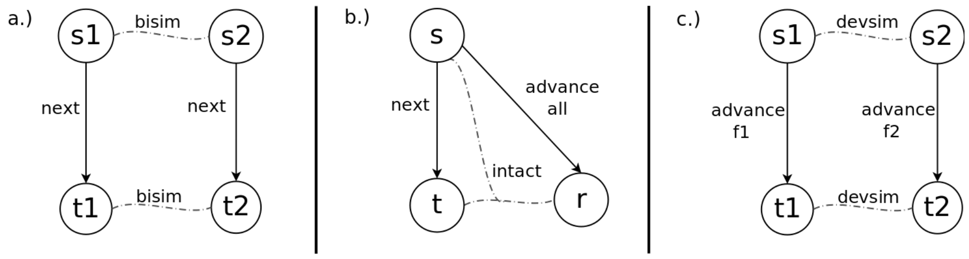

Figure D.4: a.) non-infiltration, b.) extended non-exfiltration, c.) filtered device non-infiltration

three classes, `OWN` i d, `FOREIGN` i d or `INTERRUPT` d, that correspond to the configurations `own_devices` i D, `foreign_devices` i D and `interrupt_devices` D, respectively. While configurations refer to a concrete device state D, device classes are state-independent. We require that each device is in at least one of the three classes. The system properties discussed in the following subsections have a correct configuration of the devices as a prerequisite. The configuration of each device in the current state is supposed to follow the specification of the given class. Moreover, devices are not allowed to communicate with other devices or to run into an underspecified state.

$$
\begin{aligned}
&\texttt{device\_setup}\ i\ s := \forall d.\\
&\quad (\texttt{OWN}\ i\ d \Rightarrow \texttt{own\_devices}\ i\ s.\texttt{devices}.d)\\
&\ \wedge (\texttt{FOREIGN}\ i\ d \Rightarrow \texttt{foreign\_devices}\ i\ s.\texttt{devices}.d)\\
&\ \wedge (\texttt{INTERRUPT}\ d \Rightarrow \texttt{interrupt\_devices}\ s.\texttt{devices}.d)\\
&\ \wedge \texttt{stable errorfree}\ s.\texttt{devices}.d\ \wedge\ \texttt{stable silent}\ s.\texttt{devices}.d
\end{aligned}
$$

D.6.2 Non-infiltration

Confidentiality of the kernel and neighboring user processes (including their devices) and the integrity of the active user process is guaranteed by non-infiltration, a noninterference-like property at the user mode single instruction level. Consider two machine states in user mode that are *low equivalent* in the sense that the two states agree on the resources (devices, registers and memory) that are permitted to influence user mode execution, but do not necessarily agree on other resources. Non-infiltration (Fig. D.4.a) holds if the poststates, after execution of one instruction, remain low equivalent (or produce the same error).

Theorem 12. Non-infiltration

$$\begin{aligned}
&\forall s_1,\ s_2,\ i.\ (\texttt{mode}\ s_1 = \texttt{mode}\ s_2 = \texttt{usr}) \ \wedge\ \texttt{mmu\_setup}\ i\ s_1\ \wedge\ \texttt{mmu\_setup}\ i\ s_2 \\
&\qquad \wedge\ \texttt{device\_setup}\ i\ s_1\ \wedge\ \texttt{device\_setup}\ i\ s_2\ \wedge\ \texttt{bisim}\ i\ s_1\ s_2 \\
&\quad \Rightarrow (\exists t_1,\ t_2.\ (\texttt{next}\ s_1 = \texttt{ValueState}\ ()\ t_1\ ) \wedge\ (\texttt{next}\ s_2 = \texttt{ValueState}\ ()\ t_2) \\
&\qquad \wedge\ \texttt{bisim}\ i\ t_1\ t_2)\ \vee\ (\exists e.\ (\texttt{next}\ s_1 = \texttt{Error}\ e)\ \wedge\ (\texttt{next}\ s_2 = \texttt{Error}\ e))
\end{aligned}$$

The relation bisim is the low equivalence relation. User mode processes are allowed to be influenced by the user mode registers, the memory assigned to them, devices with access to that memory, interrupt devices, the CPSR, the coprocessors, pending access violations and the misc state component. Formally:

$$\begin{aligned}
&\texttt{bisim}\ i\ s_1\ s_2 := \\
&\qquad (s_1.\texttt{counter} = s_2.\texttt{counter})\ \wedge\ (s_1.\texttt{int\_fired} = s_2.\texttt{int\_fired}) \\
&\quad \wedge\ \texttt{equal\_user\_regs}\ s_1\ s_2\ \wedge\ (\forall a.\ \texttt{accessible}\ i\ a \Rightarrow (s_1.\texttt{memory}\ a = s_2.\texttt{memory}\ a)) \\
&\quad \wedge\ (\forall d.\ \texttt{OWN}\ i\ d\ \vee \texttt{INTERRUPT}\ d \\
&\qquad \Rightarrow (s_1.\texttt{devices}.d = s_2.\texttt{devices}.d)\ \wedge\ (s_1.\texttt{ext\_out}\ d = s_2.\texttt{ext\_out}\ d)) \\
&\quad \wedge\ (s_1.\texttt{psrs(CPSR)} = s_2.\texttt{psrs(CPSR)})\ \wedge\ (s_1.\texttt{coproc.state} = s_2.\texttt{coproc.state}) \\
&\quad \wedge\ (\texttt{nav}\ s_1 = \texttt{nav}\ s_2) \wedge\ (s_1.\texttt{misc} = s_2.\texttt{misc})
\end{aligned}$$

Non-infiltration guarantees that system components outside the bisim relation can not give rise to information flow. In particular, privileged registers, memory foreign to the current process and devices that operate on such memory can not influence the execution on the CPU. External output has no impact on other components either. However, it was included into the relation to obtain guarantees on that information from the kernel and neighboring processes can not be leaked through the system's output, as long as the configuration of the devices producing that output prevents them from accessing confidential memory.

D.6.3 Extended Non-exfiltration

Non-exfiltration guarantees the integrity of resources foreign to the active user process. Given a valid configuration for user process i active, the execution of a single instruction in user mode will not modify any other resources but those considered to be modifiable by i. In [102] this was expressed by the equality of protected components in pre- and poststate. However, when some of those protected components are modified by devices executing in parallel, this equality can not be proven. Therefore, we extend non-exfiltration to a triangle shaped property (compare Fig. D.4.b), in which the poststate t of a system-wide progress is compared to both, the prestate s and a third state of comparison r that is the result of applying only the effects of the device operations to the prestate.

Theorem 13. Extended Non-exfiltration

$$\begin{aligned}
&\forall s,\ t,\ r,\ i.\ (\texttt{mode}\ s = \texttt{usr})\ \wedge\ \texttt{mmu\_setup}\ i\ s\ \wedge\ \texttt{device\_setup}\ i\ s \\
&\quad \wedge\ (\texttt{next}\ s = \texttt{ValueState}\ ()\ t)\ \wedge\ (\texttt{advance all}\ t.\texttt{counter}\ s = \texttt{ValueState}\ ()\ r) \\
&\quad \Rightarrow \texttt{intact}\ i\ s\ t\ r
\end{aligned}$$

For synchronization, `advance` is applied up to the oracle counter state in poststate t. The `intact` relation between the prestate s with active process i, the poststate t and the comparison state r guarantees that coprocessors and memory not belonging to any user process remain unchanged. The memory of neighboring user processes, new interrupts, and devices that do not access memory of i, are determined by the device operations only. In particular, they can not be influenced by writing to the memory of i. The only modifiable registers are the `CPSR`, user mode registers, and the `PSR` and the link register of the mode in t.

$$
\begin{aligned}
&\texttt{intact}\ i\ s\ t\ r :=\\
&\quad (t.\texttt{coproc} = s.\texttt{coproc})\ \wedge\ (\forall a.(\forall j.\neg\texttt{accessible}\ j\ a) \Rightarrow (t.\texttt{memory}\ a = s.\texttt{memory}\ a))\\
&\ \wedge (\forall a,\ j.\ (i \neq j)\ \wedge \texttt{accessible}\ j\ a \Rightarrow (t.\texttt{memory}\ a = r.\texttt{memory}\ a))\\
&\ \wedge (t.\texttt{int\_fired} = r.\texttt{int\_fired})\\
&\ \wedge (\forall d.\ \texttt{FOREIGN}\ i\ d\ \vee \texttt{INTERRUPT}\ d\\
&\qquad \Rightarrow (t.\texttt{devices}.d = r.\texttt{devices}.d)\ \wedge\ (t.\texttt{ext\_out}\ d = r.\texttt{ext\_out}\ d))\\
&\ \wedge (\forall q.\ q \notin \texttt{accessible\_regs}\ (\texttt{mode}\ t) \Rightarrow (t.\texttt{regs}(q) = s.\texttt{regs}(q)))\\
&\ \wedge (\forall p.\ p \notin \{\texttt{CPSR}, \texttt{spsr\_}(\texttt{mode}\ t)\} \Rightarrow (t.\texttt{psrs}(p) = s.\texttt{psrs}(p)))
\end{aligned}
$$

D.6.4 Filtered Device Non-Infiltration

In addition to the non-infiltration property of the overall system, we provide one for device activities only. It can be combined with extended non-exfiltration to guarantee that devices not accessing the active partition form their own group of resources which executes independently from the CPU. Formally, filtered device non-infiltration (Fig. D.4.c) states that devices configured to not access more than the memory of active process i (devices d for which `OWN` $i\ d$ holds) cannot influence devices not operating on that memory. Consequently, when comparing two systems and their executions, removing the activities of devices in the `OWN` class from one of the executions (through the filtering predicate of `advance`) will not change the effects that the other devices can observe.

Theorem 14. Filtered Device Non-Infiltration

$$
\begin{aligned}
&\quad f_2 = (\lambda d.\ f_1\ d \wedge\ \neg\texttt{OWN}\ i\ d)\ \wedge\ \texttt{devsim}\ i\ s_1\ s_2\\
&\wedge \texttt{device\_setup}\ i\ s_1\ \wedge\ (\texttt{advance}\ f_1\ n\ s = \texttt{ValueState}\ ()\ t_1)\\
&\wedge \texttt{device\_setup}\ i\ s_2\ \wedge\ (\texttt{advance}\ f_2\ n\ s = \texttt{ValueState}\ ()\ t_2)\\
&\Rightarrow \texttt{devsim}\ i\ t_1\ t_2
\end{aligned}
$$

The `devsim` equivalence relation describes the resources visible to interrupt devices and to devices that operate on memory of non-active user processes.

$$
\begin{aligned}
&\texttt{devsim}\ i\ s_1\ s_2 :=\\
&\quad (s_1.\texttt{counter} = s_2.\texttt{counter})\ \wedge\ (s_1.\texttt{int\_fired} = s_2.\texttt{int\_fired})\\
&\ \wedge (\forall a,\ j.\ (i \neq j)\ \wedge\ \texttt{accessible}\ j\ a \Rightarrow (s_1.\texttt{memory}\ a = s_2.\texttt{memory}\ a))\\
&\ \wedge (\forall d.\ \neg\texttt{OWN}\ i\ d \Rightarrow (s_1.\texttt{devices}.d = s_2.\texttt{devices}.d) \wedge (s_1.\texttt{ext\_out}\ d = s_2.\texttt{ext\_out}\ d))
\end{aligned}
$$

D.7 Implementation

We proved the theorems of Section D.6 for the ARMv7 platform inside HOL4. This work extends the proof presented in [102], in which we showed non-infiltration, non-exfiltration and mode switching properties for ARMv7 user mode execution on ISA level without devices. Given the complexity of the ARM model and the instruction set, we exploited automation based on a sound, but incomplete inference system. For example, for two computations f and g that both preserve non-infiltration, the inference rule for sequential composition derives that also $f \gg_{\texttt{nav}} g$ preserves non-infiltration. We have proven further rules for parallel composition, loops, alternatives, lambda abstraction and other constructors of the operational semantics. They enabled us to develop a proof tool for relational and invariant reasoning that - after being provided with the desired properties for primitive operations - was able to discharge large parts of the proof obligations (but not all) automatically. Details are discussed in [102].

In the present extended work, the separation properties had to be proven manually for `advance`, mainly because they would not hold for intermediate computations in isolation. Due to the complexity, this was one of the main challenges. We followed a bottom-up approach. Basic properties on `mem_acc_by_dev`, a rather extensive case analysis and automatic simplification allowed for the verification of properties on `advance_single`. This step often required to split the analysis into the effects on the device currently progressed by `advance_single` and the effects on all other devices. Finally, properties for `advance` were proven by induction. In order to allow for the continued application of the proof tool to the existing parts, we had to verify the transitivity of `advance`. Subsequently, the vast majority of the automatic proofs could be repeated without any interruptions, which gives confidence that our proof framework scales well for extensions of the platform model. The Cambridge model of ARM is 9 kLOC. In addition to the ARM model, we rely mainly on the relatively small inference kernel of the HOL4 theorem prover, our MMU extension (about 180 lines of definitions), the device framework (about 350 lines) and the formulation of the discussed properties (about 380 lines). The entire proof script has a length of about 20 kLOC and needs roughly two and a half hours to run on an Intel(R) Xeon(R) X3470 core at 2.9 GHz. We invested about five person months of effort into this work.

D.8 Conclusions

Summary We extended the Cambridge HOL4 ISA model for ARM by a general device framework for DMA devices. Based on the extended model we identified secure device configurations and proved several isolation properties for platforms where DMA devices execute in parallel with a CPU in user mode. The results can be used in separation proofs, be it in a hypervisor, separation kernel or operating system setting. Model, properties and verification approach can be adapted to other architectures. We gained confidence that our proof framework scales well for extensions of the model.

Future Work The model allows for further interesting angles, which we plan to explore in future work: It is rather common that devices communicate with each other. So far, we can only support such constellations by merging communicating devices into one block, so that the model understands the block as a single device. Removing this restriction comes with the challenge of ensuring that device configurations still remain secure when devices are allowed to write to ports of other devices. Probably easier to achieve is the augmentation of the set of device classes by devices that neither use DMA nor interrupts, but that can be accessed by user space processes. A UART interface managed by a single process is one such example. From a security perspective, such a device is similar to physical memory assigned to a process, in spite of the self-modifying nature and external influence that such components have. Even if devices (like a timer) are shared between different user processes, user mode access to their ports can still preserve isolation, for example, if that access is always reading. Further potential future work includes the investigation of connected external input/output channels or the enhancement of the model by an IOMMU.

Discussion on Weak Memory Models This paper focuses on secure device configurations and the integration of devices into the platform information flow analysis on ISA level. We have deliberately chosen a flat sequential memory model and leave weak memory effects for future work. In the following, we briefly discuss possible benefits of such future effort. A sequential memory model does not accurately account for all observable effects that can occur in practice. Even in a single core system with peripheral devices, system components such as store buffers or caches can cause behaviour that is not reflected by our current model. Furthermore, practical implementations of bus and peripherals might violate assumptions we made in this paper. For instance, in Section D.5 we assumed that read requests sent by devices to the bus are answered in an atomic manner. For some bus systems this assumption holds, for others it does not. In practice, most devices are not affected by this matter, since they wait for read request being answered before issuing new ones. However, for some combinations of peripherals and bus systems our current model might miss some behaviour. This gets apparent when multiple reads are considered: assume a zero-initialized memory and two locations that are overwritten with 1's by the CPU. Without loss of generality, let the first write that takes effect be at location a and the second write be at location b. A peripheral might issue a read request for b first and then – without waiting for the response – a read request for a directly afterwards. If the request for a is answered before the one for b, then the peripheral will not only notice the inverse order on a low level, it may also receive the values $b = 1$ and $a = 0$. In our current model there is no oracle instantiation that would cause this outcome for reading request order b before a.

More accurate models might be beneficial for the functional verification of drivers, for example. When it comes to the information flow properties studied in this paper, the implications of a sequential or a weak memory model are more subtle. A potential non-atomicity in the handling of memory read requests from a device could conceivably lead to reply orders that depend on the activities of other devices. This might allow a partition to infer if devices of other partitions are currently operating on the bus. It is not clear how much exactly a partition might be able to learn about others through such a channel. Reply orders depending on otherwise inaccessible registers or memory locations could allow an attacker to infer information about these resources, potentially even in a more controllable and predictable manner than with the channel discussed above. However, a dependency of the read order from such system components is not documented in the ISA specifications. They do not rule such dependencies out either, but we believe that for non-malicious hardware implementations their existence is unlikely, and certainly not something that is intended by the ISA designers. A model that does not assume atomicity in the reads, but applies a completely non-deterministic read order cannot uncover any additional flows and would in the end – like our current model – eliminate such information channels that are not supported by explicit descriptions of dependencies in the ISA specifications. Developers who write drivers that are dependent on a specific reply order or who need to ensure confidentiality of device actions beyond logical level are advised to take measures external to the model.

In a context switch, the kernel usually sets up new address translations and access control policies. Directly after returning to user-mode, weak memory might lead to the usage of unintended addresses or access rights. However, kernels can employ cache evictions and barriers to ensure that static access permission settings take effect before execution is handed over to unprivileged processes. Side-channels related to caches [83] have been widely studied. Not all of the cache issues discussed in literature apply to our scenario, but caches are relevant even for systems with a single core and devices. Typically, an attacker partition might attempt to use the cache to learn about the control flow of a victim partition and, for example, infer secret keys [185]. To that end, the attacker might fill parts of the cache, wait until it is called again after the victim's execution phase, and check which cache lines have been evicted. The last step is not trivial, since the attacker cannot directly see whether a value in a certain address resides in cache or the actual memory. But if the attacker can share cacheable memory with one of its devices, then it might be able to configure the device in such a way that it copies the address to observe to some other shared (non-cached) memory location. Since the device reads the values from memory and not from the cache, the attacker would learn whether the value it has written to the cache is still there or has been evicted. A kernel can counter this attack by disallowing the shared memory between a partition and a device to be cacheable. To that end, the kernel would need to know the exact range of the DMA operations or over-approximate the range with possible performance costs.

The discussed issues are examples of the relevance of weak memory models. The list is not complete. In particular, we have not discussed timing channels, which are hard to avoid and would need models of a different nature than the one presented in this paper. Generally, more work needs to be done to understand the implications of weak memory effects for ISA security analyses such as the one we introduce in this paper. The creation of sufficiently precise models is a challenge in its own right.

Paper E

Securing DMA through Virtualization

Oliver Schwarz and Christian Gehrmann

Abstract

We present a solution for preventing guests in a virtualized system from using direct memory access (DMA) to access memory regions of other guests. The principles we suggest, and that we also have implemented, are purely based on software and standard hardware. No additional virtualization hardware such as an I/O Memory Management Unit (IOMMU) is needed. Instead, the protection of the DMA controller is realized with means of a common ARM MMU only. Overhead occurs only in pre- and postprocessing of DMA transfers and is limited to a few microseconds. The solution was designed with focus on security and the abstract concept of the approach was formally verified.

E.1 Introduction

Different types of embedded systems are present in all kinds of computing and control devices ranging from low power sensor nodes to mobile phones. Not only the usage of embedded systems is increasing, also the number of their vulnerabilities is. The trend towards the usage of common software components, highly interconnected systems and open architectures increases the risk of software attacks even more, evoking the need for more powerful protection mechanisms. A promising approach is the usage of virtualization as mean to isolate security-critical from non-security-critical execution, for example a commodity operating system from trusted services. Such a separation can be achieved independently from the complexity of the guest systems. In fact, the trusted computing base can be reduced by the introduction of a virtualization layer. The smaller the trusted computing base is, the earlier it is possible to get assurance on the security achieved, for example through formal verification. The idea of using virtualization as an enabler for security is not new; however, there is still a deficit of solutions targeting the special needs of embedded systems while keeping a small code base. In [75] we presented such an approach. It targets the widespread ARMv5 platforms and focuses on security by providing isolation between guests. With a footprint of less than 10 KB the described hypervisor is reasonable thin. Furthermore, the performance overhead has been shown to be low, even though ARMv5 does not provide any advanced hardware support for virtualization. This shortcoming is expected to change in the future and the recent ARMv7 Cortex-A15 has additional hardware support to handle virtual to physical address translation [14]. Their System MMU [120] even targets *Direct Memory Access (DMA)* in particular. However, legacy embedded systems will exist for a very long time and currently there is still a lack of essential hardware functions such as assisted handling of peripheral units by an IOMMU [2].

To provide peripheral devices with quick access to the memory, DMA became essential even in embedded systems. DMA controllers are able to copy data faster than the CPU. However, most of the related work that aims at virtualization of DMA does not (or not sufficiently) take care of security matters in respect to DMA. The purpose of our paper is thus to address this issue. That is indeed needed as DMA potentially opens an alternative way for attackers to modify otherwise inaccessible memory by bypassing CPU and MMU. We show how to extend the previous hypervisor design by adding protection against such attacks. Common solutions use to address this challenge by the usage of an IOMMU. However, many embedded systems lack this hardware feature. We therefore show how to prevent DMA attacks with high secure isolation guarantees through DMA virtualization based on standard ARM *Memory Management Unit (MMU)* functionality only. Our design approach is generic and applies to newer ARM architectures such as ARMv6 and ARMv7 as well. Embedded platforms without hardware virtualization support like ARMv5 will remain widespread for many years. Especially low end devices still use early designs.

The following contributions are made in the paper:

- We describe how virtualization can be used to protect DMA on ARMv5 based systems.
- We show that secure DMA virtualization is feasible without additional hardware.
- We make a careful security and performance analysis of our design proposal.
- We provide a formal proof of the abstract concept of the approach.

The remainder of this paper is organized as follows: First, we give an overview of related work. Next, assumptions, the threat model and security requirements are defined. After an introduction to the ARM architecture and the hypervisor, the DMA virtualization approach is explained in detail. An analysis of the results and a discussion of the concept's formal verification subsequently underline the value of the approach. Finally, we conclude the paper.

E.2 Related Work

Virtualization as mean to provide security in embedded systems was discussed in [42]. Previous attempts to actual use virtualization on embedded systems have mostly been focused on porting of widely used virtualization layers such as Xen [24], [98] or on performance analysis aspects [18].

That strong isolation can be achieved through advanced combination of virtualization and the standard memory protection support on for example Intel based systems was convincingly showed in the recent works by Seshadri et al in [155] and [115]. Our design allows even the support of DMA without loss of isolation or the need for advanced I/O memory protection (such as an IOMMU). Härtig et al. [86] describe how to monitor potentially malicious device drivers of a system without a hardware IOMMU. However, different from our approach, their work is neither designed to support several guests nor does it cover scheduling or virtual views on the DMA controller. Furthermore we put the assurance of security properties into the focus.

Recent work has shown that verification of low level software is in fact feasible. Klein et al. successfully verified the micro kernel seL4 [105] and the Robin project [171] made progress on the verification of the Nova hypervisor [168]. This gives us motivation to perform verification effort on our solutions (such as the here described approach) as well.

E.3 Prerequisites

E.3.1 Assumptions

The following assumptions are made in this paper:

- The only DMA controller present on the platform is a general purpose DMAC not bounded to any particular peripheral. It does not perform any action without being programmed to do so by the CPU. The programming interface of the DMAC is known to the hypervisor.
- The MMU supports the management of ARM domains (see Sect. E.4.2). Peripherals are memory mapped and access to them can be controlled through the MMU.
- All hardware entities, including the DMAC and the MMU, are working correctly and are not malicious. No physical attacks or tampering attempts are present.

- The hypervisor code is loaded unmodified at system startup. All upstream entities (such as the BIOS and the boot loader) are trusted. To ensure this, a secure boot process [10] can be used.

E.3.2 Threat Model

The attacker is assumed to have complete control over one or several guests. That includes the possibility to execute arbitrary code and to access data with its/their rights. The attacker's goal is to compromise any other guest, that is, to modify or read its code or data, or to prevent, delay or control execution of foreign code. We assume that he or she has no intention to determine whether DMA is used by other parties or not. However, should a stricter security property be desired, the hypervisor could be extended in one of the following ways to prevent revealing delays:

- "Secret" DMA tasks can be stopped in favor of DMA tasks issued by a guest which might possibly be interested in observing the DMA activities of the system. They can be resend to the DMAC after other tasks are done.
- The hypervisor can introduce pseudo delays if the DMAC is not used by other guests, so that an attacker cannot distinguish between free and occupied DMAC channels.
- The concurrent execution of guests can be (temporarily) disallowed or the hypervisor can wait until all "secret" DMA tasks are finished before a potentially malicious guest is called.

All those measures can be provided optionally so that they only apply for certain critical guest configurations.

E.3.3 Security Requirements

The goal is to provide isolation between guests to prevent any of the attacks described in the threat model. Furthermore it shall be guaranteed that DMA functionality is always available to non-malicious guests. This means that any policy-conform DMA task will eventually be processed. Given those targets, the following security requirements need to be implemented:

1. Every attempt to access the DMA controller leads to a trap into the hypervisor.
2. The DMAC performs only those copy operations which comply with the access policy. A word is accessible according to that policy if and only if it can be written/read by the current guest even without DMA support, that is, via the CPU, but in context of the valid MMU settings.

3. No guest can (re-)program a DMA request on behalf of another guest. This is independent from the access policy. Assume guest 1 has access to the addresses A, B, C, D and intents a copying from A to B, then guest 2 is not able to alter this request to, e.g. “from C to D”, even though this would be a valid request when issued by guest 1 without interference of guest 2.

4. The scheduling does not influence the security.

5. Virtualization solutions which are supposed to provide security may have the vulnerability that they are attackable in their configuration [151]. In contrast, our hypervisor can not be modified by guests, neither using DMA functionality nor otherwise.

6. All DMA tasks that comply with the policy are either directly processed by the DMAC or enqueued.

7. Every DMA request will be processed (either executed or denied) and subsequently deleted from the DMAC and internal structures, so that they can not get blocked.

The isolation target is covered by requirements 1 - 5. DMA functionality can only be used through the hypervisor (1), which itself is not modifiable (5) and works correctly. Working correctly here means that it only grants valid accesses (2) issued by the right guest (3) and that itself will not modify accesses so that they would influence security (4). Availability follows directly from requirements 6 and 7.

E.4 Architecture and Hypervisor

E.4.1 Operation Modes

ARMv5 has one non-privileged mode and six privileged modes without further hierarchy. As a hypervisor needs to supervise guest kernels, it has to operate in the privileged ring while all guests have to be placed completely into the non-privileged mode. On interrupts and data aborts, the CPU switches to the privileged ring. By using a software interrupt, guests can intentionally pass control to the hypervisor.

E.4.2 Memory Access Control

The MMU is used to ensure separation between guests and to protect the hypervisor. The ARM architecture allows to define so called “domains”. Each page can be assigned to one of them. Depending on the access bits for a domain, all its pages are either not accessible at all, fully accessible or subject to the settings on page table level. This allows the simultaneous changing of memory access rights for all domains by only one register access. Input and output devices are memory mapped and are thus also subject to the MMU based access control. This enables a hypervisor to intercept on interaction with the DMAC via an according abort handler.

E.4.3 OVP and the DMA Controller

We have developed a proof of concept implementation, which has been tested on an emulated platform by *Open Virtual Platforms (OVP)* [133]. Our implementation utilizes the general purpose DMAC provided by OVP. Different from the ARM provided *PrimeCell DMA Controller (PL080)* [16], the OVP DMAC has only two channels instead of eight and is restricted to the simplest functionality. However, the programming interface is similar and the simplified DMAC is still sufficient to demonstrate the effectiveness of the approach described in this paper. The focus lies on four channel specific registers, namely a source and a destination register, the control register, in which information about the burst size and the total size of the data is encoded, and the configuration register, which causes the copying to start on certain values.

E.4.4 The Hypervisor

The starting point for our design was a hypervisor previously designed and implemented by the Swedish Institute of Computer Science. The hypervisor aims at the isolation of guests. Each guest is made up of an arbitrary number of guest modes. Besides separating different guests from each other, the hypervisor can also strengthen isolation between the guest modes of one single guest (such as the user and the kernel mode of an operating system). In either case the hypervisor can be configured to allow inter-mode-access mono- or bi-directional or to prohibit it so that isolation is guaranteed. A typical scenario would include a setup of one commodity operating system along with a domain (guest mode) for trusted services. Several memory regions can be defined, for example one for each guest mode, and pages can be assigned accordingly to determined ARM memory domains, which allow quick locking and unlocking of the guests' data and executable code. Additionally, when switching between two guest modes, the hypervisor saves and restores the respective contexts, that is, the registers' contents. For the communication between guest modes the hypervisor offers the possibility of establishing remote procedure calls (RPC). The operating system FreeRTOS was successfully ported to the hypervisor.

E.5 DMA Virtualization

Common to other hardware virtualization solutions, we use an approach where we emulate the DMAC. Different from typical approaches though, our design is driven by a careful security analysis, making sure that a hostile guest will not be able to circumvent any system access rules. This includes interrupt handling and memory access control. Guests do not interact directly with the physical controller.

Instead, each access attempt will result in trapping into the hypervisor, which then controls and manages the DMA tasks before forwarding them to the physical DMA controller. Thus, programming a DMA task appears to the guest as if there were no virtualization. In the background the hypervisor checks access conditions and takes care of scheduling issues. When the DMA task is done, the hypervisor forwards the interrupt it has received from the DMAC to the guest in charge.

E.5.1 Shadow Copies and Scheduling

To prevent guests from interfering during foreign DMAC setups, the hypervisor maintains shadow copies of the DMAC (one for each guest mode). When a guest tries to access a certain register of the physical DMAC, the hypervisor is invoked via the data abort handler and writes the given value into the guest mode's shadow DMAC instead. The physical DMAC will only receive complete and bundled data from the hypervisor. There might exist more guests concurrently interested in DMA than resources are available. As guests may assume that they are possessing the hardware for their own use, they will not actively wait with submitting DMA requests until the DMAC is not longer occupied. Instead, they will post their request assuming that it is processed. Thus, the hypervisor keeps track on which guests have commanded a DMA task and memorizes the parameters as long as the task is not sent to the real DMAC (when occupied). The decision to manage shadow DMACs makes the memorization of parameters simple. A queue is used to schedule the DMA tasks. As we assume a symmetric system, tasks can be assigned to any free channel of the physical DMAC, regardless of whether the physical and the virtual channel numbers are equivalent. Depending on whether a DMA task is enqueued or posted to the physical DMAC, the according virtual channel in the guests's shadow DMAC will be marked as *waiting* or *active*, respectively.

E.5.2 Trapping with the Data Abort Handler

After trapping to the privileged ring, the data abort handler of the hypervisor determines from where an access attempt came and to which address it was directed. To get to know whether it was a reading or writing access and which CPU register was supposed to be involved, the hypervisor decodes the machine instruction. If the instruction refers to the DMAC, the hypervisor will calculate which channel and which register was meant and as long as the (virtual) channel is not marked busy the value to be written will be filled in into the shadow register. In case of an correctly formatted access to the configuration register, the DMA task in question is checked with respect to the access policy.

E.5.3 Access Control

The DMA access policy is directly derived from the system's access definitions. Whether or not a guest is allowed to read or write from or to a certain spot in memory is already defined in the MMU coprocessor registers and the access control data structures of the hypervisor. The same rules will apply for copying operations with the use of DMA.

E.5.4 Handling DMA Interrupts

The purpose of DMA interrupts is to inform the issuer of a DMA task that the operation is done. In our scenario the hypervisor will receive those interrupts. Besides forwarding them to the corresponding guest mode by calling a special handler provided by the guest, the hypervisor also uses the interrupts to determine when the DMAC is available again. Summarized, the hypervisor safes the execution context of involved guest modes, disables further interrupts and switches to the interrupt handling mode. After finishing its procedure, the just called guest yields back to the hypervisor by a hypercall. Finally, the hypervisor re-enables interrupts, dequeues waiting DMA tasks and returns to the interrupted guest mode.

E.6 Evaluation

E.6.1 Performance

With the help of the emulator by OVP we compared the performance of the described virtualization solution with DMA support to:

1. the performance of DMA in a non-virtualized (not protected) system and

2. the performance of copying without DMA.

Especially the latter underlines the value of our work. The usage of an emulator allows us to make a very detailed performance analysis based on the granularity of cycle numbers. Table E.1 gives an overview on how many cycles the setup of the DMAC, the programming of a task and the handling of a DMA interrupt take in the standard and the secured virtualized version, respectively. The DMA programming step is detailed analyzed in respect to its substeps: trapping into the hypervisor, obtaining the shadow copy (SC) of the guest mode, filling out the shadow copy, performing the access control check, submitting the task to the real controller and the possible need to enqueue a task. The number of required cycles of different requests can vary slightly, depending on the channel addressed (due to address calculation), the number of channels used, the access policy etc. Therefore, both the minimum and the maximum values of our tests are listed. The upper bounds for setup, programing and interrupt handling, respectively, are compared to the performance values of the non-virtualized reference system. The resulting factors are listed in the fourth column. Our reference processor, the ARM926EJ-S, performs at least 200 million instructions per second (MIPS) so that the required time for the single steps can be approximated (in micro seconds). Even though the factors seem to be quite high at first glance, the security effort is nonetheless reasonable for a solution that does not require any hardware changes to the legacy system at all: a whole DMA cycle (programming a task and handling the interrupt) will not take more than 15 micro seconds. This low overhead in the pre- and postprocessing is negligible compared to the benefits of using DMA (for details see below).

Table E.1: Overhead of securing through virtualization

step	standard	virtualized	×	μsec
setup of DMAC	51	51	1	0.26
programming				
.. trapping		466		
.. obtaining SC		135 - 178		
.. filling out SC		233 - 272		
.. access control		462		
.. submission	44	380 - 458		
.. (enqueuing)		(57)		
= total	44	1676 - 1836	42	9.18
interrupt handling	89	1155	13	5.78
total per task	133	2831 - 2991	23	14.96

Besides comparing non-secured and secured DMA with each other, we also have analyzed when the performance benefits of secure DMA exceed its costs. More specificly, we have measured the CPU cycles required to copy different amounts of data without DMA, that is, only with the CPU running copying instructions in a loop. Figure E.1 displays those results ("CPU") and compares them to the constant costs of using DMA ("DMA"). Copying a single word (4 bytes) without DMA requires 21 to 25 cycles on average, depending on the data structure and loop overheads. While for small tasks the use of DMA is not appropriate, from around 128 words (0.5 KB) on it consumes less CPU cycles than the standard way. Note that both variants are secured by the hypervisor. Finally, the graph labeled "DMA/CPU" shows the ratio of the costs when copying with DMA and CPU only, respectively. It is clearly visible that the use of DMA quickly becomes strongly preferable, even without loss of security.

E.6.2 Security Analysis

Referring back to the security requirements defined in Sect. E.3.3, we now go through them step by step (original numbering kept) and reason why our solution fulfills them.

1. The memory region in which the DMAC is situated is configured to deny any access from the unprivileged processor mode in which all guest code is

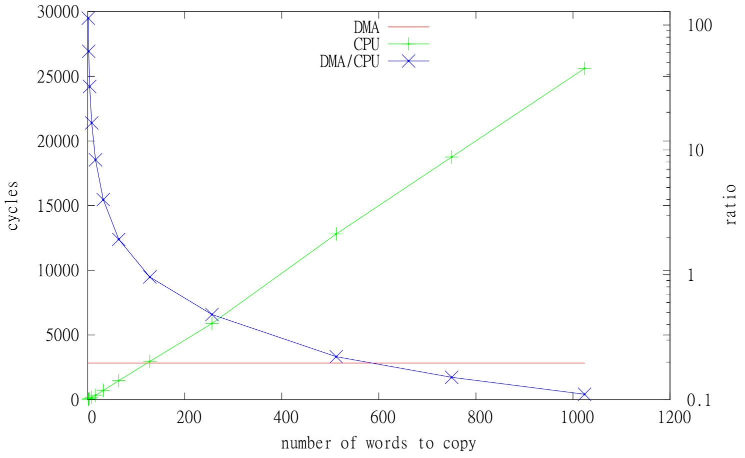

Figure E.1: Comparison between DMA and CPU copying of words in a secured environment

executed. Every access attempt invokes the data abort handler defined by the hypervisor.

2. The hypervisor is the only entity which is able to program the DMAC. Before filling out its registers, the DMA task in question is checked according to the same criteria which apply to writing and reading processes without DMA support. Should this test fail, the task will neither be submitted nor enqueued. Once a task is actually submitted or enqueued, it cannot be modified anymore.

3. All programming steps are performed on a guest specific shadow DMAC first. That is, no guest can interrupt another one and modify specifications such as source and destination address in the name of the interrupted guest. This protection ensures that interferences during the programming of the DMAC are not possible. Only the very last step, the actual commitment, causes the physical DMAC to be filled out. This is done by the hypervisor on the basis of the shadow DMAC of the current guest. With other words, no data from other guests can influence this process. Interrupts are disabled during this last step.

4. The queuing operations do not modify the actual content of DMA tasks. Only the physical channel chosen can differ from the virtual channel number, which does neither affect which data is copied nor to which address. Furthermore, the hypervisor keeps track from which guest a certain DMA task came so that interrupts are forwarded to the actual issuer of a DMA request.

5. The memory region in which the hypervisor is situated is configured to deny any access from the unprivileged processor mode, in which all guest code is executed. Therefore, the protection of the hypervisor itself follows from observation 2.

6. The number of bytes to copy and the source and destination address of a DMA request communicated by the guest are saved without modifications in the shadow structure of the guest mode. As each guest mode has an own shadow copy, no interference between guests can occur. The request procedure is completed by a write attempt to the configuration register. This will invoke the access policy check. In case of acceptance, the task is submitted to the DMAC or, if this is occupied, will be enqueued. The queue can not overflow as the number of DMA tasks marked waiting is limited by the number of guest modes times the number of DMA channels. This follows from the fact that each channel can only be programmed once at a time per guest mode.

7. First, the characteristics of a queue ensures that each task enqueued will be processed at some time. Invalid requests will not even be enqueued. Tasks are finite and after their completion the DMAC will fire an interrupt, which will reach the hypervisor. The latter will delete the task from the DMAC and its internal structures. If an interrupt occurs while another one is processed, the hypervisor will notice the second task completion as well as it (re-)checks the states of all DMAC channels immediately before leaving the interrupt service routine again. However, denial of service attacks can be a threat to the complete handling of DMA interrupts. How they can be prevented or at least limited is described below.

The possibility that a guest does not yield back to the hypervisor is always given, not only in the context of DMA support. A good way to get control back in any case is the use of a timer tick counter or watchdog interrupt, which will call a hypervisor function after a predefined amount of time. Especially for the DMA interrupt handling presented here there are additional means by extending the configuration of the hypervisor. For example, the hypervisor can be changed to disallow or postpone the interrupting of guest modes which are seen as especially important and protect worthy. That way, a malicious guest mode cannot use a DMA task to get execution time (via its DMA interrupt handler) during a sensitive operation of another guest mode. Vice versa, it is conceivable to grant only trusted guests the possibility of own DMA interrupt handlers, at least in critical situations. But not only the DMA interrupts need to be considered. Each attempt to program the DMAC, no matter if successful or not, causes a delay in the system. To prevent malicious guests to use this for slowing the system down, the hypervisor can easily be modified to restrict accesses to a certain number or frequency per guest mode.

E.7 Formal Verification of a Simplified DMA Model

One of the benefits of using virtualization for security is the relatively small size of a special purpose hypervisor compared to a complex modern operating system. This reduces the trusted computing base distinctly and allows formal verification of the system's overall security properties. We used the *Coq* theorem prover [33] to show memory isolation in the context of DMA on a highly simplified model.

Assuming there are exactly two guests, either of them potentially malicious. Guest 2 is the "object of comparison": it is shown that the memory region and data structures assigned to guest 2 are not influenced by DMA operations of guest 1. It follows that both the integrity of guest 2 and the confidentiality of guest 1 hold.[1] We model the machine state as a record of a simplified memory, the DMAC and its two shadow copies, flags indicating which guest system has DMA transfers active, resp. waiting, and finally an interrupt flag indicating the completion of a DMA transfer. We assume that a guest system can read the memory addresses belonging to it at any time. In contrast, the MMU prevents accesses (not using DMA) to the other guest's memory. An execution consists of a sequence of state transformations. Three state transformations are represented in the model, namely the activities of the hypervisor initialized whenever a guest attempts to access the DMAC, the operations performed by the DMAC and the functionality of the interrupt handler. As those transformations usually occur together, we summarize such a chain to an execution block, denoted by F. Depending whether it was evoked by guest 1 or 2, it is referred to by F_1 or F_2, respectively. Following those conventions the execution of the system can be seen as a sequence of blocks, as for example $F_1 - F_2 - F_1 - F_2 - F_2 - F_2 - F_1 - \ldots$. We show that, in this model, under no circumstances the content of the memory region for guest 2 will depend on the inputs made by guest 1. With other words no information can flow from guest 1 to guest 2. This is done by proving that for any execution sequence, neither the memory content belonging to guest 2 nor the content of guest 2's shadow DMAC registers change when eliminating blocks of type F_1. The proof uses an important state invariant, reflecting that the state is *correctly and soundly configured*, defined as follows:

- The DMAC is set up according to the access policy.
- If some shadow DMAC is marked enabled, its source and destination registers follow the access policy.
- No guest is active and waiting at the same time.
- The interrupt flag is only set if the DMAC is enabled.
- The physical DMAC is enabled if and only if at least one of the shadow DMAC is.

[1]Confidentiality of guest 1 and integrity of guest 2 hold by symmetry.

- A shadow DMAC is marked enabled if and only if the corresponding guest is marked as *active* or *waiting.*
- If some guest is marked as *waiting* then the other guest is marked as *active.*

It is shown that all hardware and hypervisor operations maintain this invariant, that is, when starting in a correct and soundly configured state, the execution will result in another correct and soundly configured state. Finally the main theorem is formulated as follows: Starting in a correct and sound configured initial state and building up two execution sequences of which one represents any actual execution and the other is the related sequence ignoring all operation/execution blocks caused by guest 1, then at every step both sequences will be equivalent with respect to guest 2. We were successful in proving the theorem in Coq using 2200 lines of proof, out of which 300 describe the actual model. Even though this initial verification effort demonstrates that the DMA approach of the paper is based on sound fundamentals, it still relies on strong assumptions and a very simplified model. Work on refining the approach is in progress.

E.8 Conclusions

We have demonstrated that DMA virtualization only based on software and standard hardware can provide high security guarantees. To strengthen this even more we also have applied formal verification. Besides isolation, availability is warranted. Moreover, our approach has a low performance overhead, on the order of a few microseconds in pre- and postprocessing of DMA transfers. We intend to compare our performance results to systems with ARMv7 and/or System MMU. Additional features such as broader platform support or secure booting are also left for future work. As for verification, we plan to analyze the whole hypervisor on binary level.

Paper F

Affordable Separation on Embedded Platforms: Soft Reboot Enabled Virtualization on a Dual Mode System

Oliver Schwarz, Christian Gehrmann, and Viktor Do

Abstract

While security has become important in embedded systems, commodity operating systems often fail in effectively separating processes, mainly due to a too large trusted computing base. System virtualization can establish isolation already with a small code base, but many existing embedded CPU architectures have very limited virtualization hardware support, so that the performance impact is often non-negligible. Targeting both security and performance, we investigate an approach in which a few minor hardware additions together with virtualization offer protected execution in embedded systems while still allowing non-virtualized execution when secure services are not needed. Benchmarks of a prototype implementation on an emulated ARM Cortex A8 platform confirm that switching between those two execution forms can be done efficiently.

F.1 Introduction

Embedded systems are becoming more powerful, distributed and globally connected. We see a transition from classical single function embedded systems to powerful collaborative special purpose computing devices often controlling sensitive or critical infrastructure functions, so called *cyber-physical systems*. In the past, software attacks were mainly targeting high performance computers such as desktop computers, laptops, and recently also mobile devices. This is about to change rapidly. Security threats against cyber-physical systems have become a severe issue, requiring strong platform security protection techniques such as separation [158] without overly increasing performance or system costs.

The need for separation of security critical data and code on mobile devices motivated ARM to introduce the TrustZone technology [11], available for some (but not all) ARM systems. TrustZone is a System-on-Chip (SoC) isolation technique that establishes a high degree of separation between *trusted* and *non-trusted* execution, while keeping context switches fast. To distinguish between trusted and non-trusted address space, TrustZone adds an additional address bit to the bus system. In order to not break isolation, careful SoC adaptations at the design level of application specific integrated circuits (ASIC) are necessary to make memory interfaces, interrupt controllers, Direct Memory Access (DMA) devices etc. aware of that bit.

System virtualization is an alternative way to protect security critical assets [79, 80]. However, in tiny embedded systems with limited hardware virtualization support, system virtualization implies a non-negligible performance overhead [58]. On the other hand, security services typically do not run on the system all the time. They can be scheduled on a regular basis to perform monitoring or be called upon demand (e.g., for secret key operations).

In this paper we propose an alternative system virtualization enabled approach for separation, based on dual mode execution, i.e., the ability of choosing between virtualized and non-virtualized execution mode, and switching between the modes through soft reboots. The goal of the solution is to provide separation while keeping both performance overhead and required SoC adaptations to a minimum. Only a few hardware adaptations to an existing architecture are required. In one of the typical use cases, a service for proving the device's identity to its environment wants to keep the authentication key secret from the rest of the system. The system would run non-virtualized in the majority of the time, but activate the trusted service domain only for the actual authentication process. The exchange of required challenge-response-messages throughout that process will happen via remote procedure calls (RPCs).

Different from general purpose hypervisors (also called virtual machine monitors (VMM)) such as Xen [98] and KVM for ARM [58], a hypervisor with the purpose of separation or monitoring has a more focused scope and several optimizations can be made. We have developed a tiny hypervisor for ARM Cortex A8 with focus on separation. It was recently released as open source, and isolation properties of one version of this hypervisor have been formally verified on binary level. Based on this hypervisor, FreeRTOS as main guest, and emulated ARM Cortex A8 hardware enriched by our hardware extensions, we have implemented the suggested approach for dual mode protected execution. Benchmark figures show the feasibility of the concept. The main costs for enabling isolated services consists of their decryption and the integrity check of those services and of the lightweight hypervisor. Returning to non-virtualized execution does not take much longer than the erasure of newly produced confidential data.

Contrary to other approaches, that are for example based on TrustZone or trusted computing enabled late launch [82], the solution presented in this paper does not require any particular CPU architecture or extensions to the CPU, which keeps costs low and makes the concept applicable to a large set of embedded systems. Summarized, our solution offers the following benefits:

1. Trusted domains can be executed with guaranteed separation without causing performance overhead in phases where their services are not required.

2. If desired and the use case allows the resulting latency, the commodity OS can be paused throughout the protected phase, so that trusted domains can execute without the need of paravirtualization[1] of the commodity OS.[2]

3. The proposed protocol includes a secure boot scheme, so that confidentiality and integrity of hypervisor and trusted domains are maintained even in the presence of external accesses to their non-volatile storage.

F.2 Hardware and Protocol

We consider a concept that relies on minor adaptations on SoC design level to make it possible to run the system in two modes, *protected mode* and *normal mode.* In protected mode a dedicated hypervisor runs in the most privileged level on each CPU in the system and trusted guests (such as secret key services) can run separated by the commodity OS, while in normal mode no hypervisor needs to be present in the system, as depicted in Figure F.1 for a single CPU system.[3] Priviliged software can cause transitions between modes by requesting a *soft reboot* (also referred to as *soft reset* or *warm reboot*), which is initiated by the system's reset signal.

The SoC contains two special purpose *volatile* memory registers: a *mode state register* and a *transition register.* The mode state register states whether the system is currently in protected or normal mode. The transition register is used to state the intention of commodity OS or hypervisor about which mode to enter. The mode state register can *only* be changed in early booting phases. Thereafter it will be locked through a sticky bit so that it can not be modified anymore until a chip reset (and consequently a soft reboot) occurs. The boot code responsible

[1]Paravirtualization [165, p. 422] describes any modification of guest operating systems, in order to enable their execution on a virtualized environment instead of bare metal, e.g. by making them use software interrupts (*hypercalls*) to perform privileged operations, according to the hypervisor's API.

[2]Depending on the scenario, interrupts would be recorded by the hypervisor or just masked during the pause.

[3]Here, we illustrate a single CPU architecture, but the principle can easily be extended to a multicore architecture, see Section F.2.1.

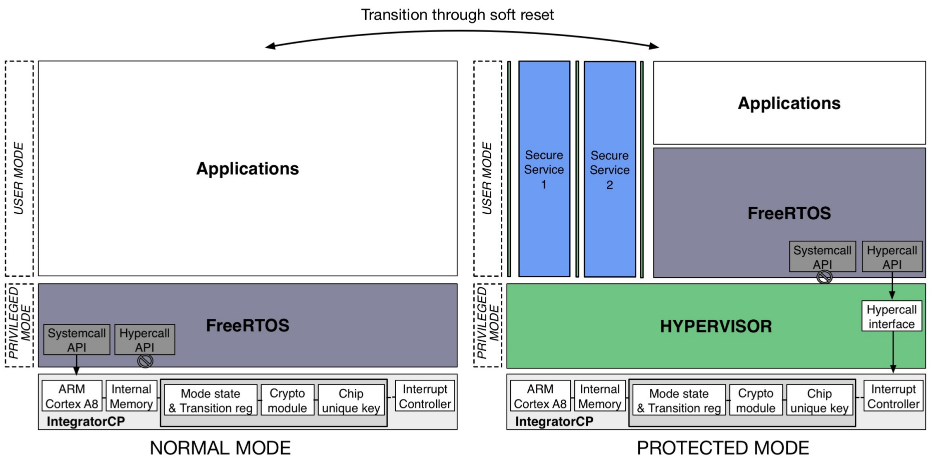

Figure F.1: Dual mode operation.

for the hypervisor and OS kernel launch determines which mode to boot into - and consequently the value to set in the mode state register. In a *cold boot* (full hardware reset) the default mode value is given by a boot configuration. In a warm/soft reboot the value is determined by the transition register, as set by the higher level software.

When running in protected mode, the hypervisor controls sensitive applications, I/O devices and data and can protect the system from illegal access to these units. This can be achieved using the normal Memory Management Unit (MMU) or Memory Protection Unit (MPU) present in most systems. If applicable, additional hardware protection support can be utilized, such as an Input/Output MMU (IOMMU). The memory protection mechanisms are also used to make sure that, when running in protected mode, a soft reboot to normal mode can *only* be initiated by the hypervisor or hypervisor protected units, such as a watch dog timer reset function (placed in a protected address space). [4]

Figure F.2 shows a SoC design according to the approach and the proof of concept implementation we have done using emulated hardware (see Section F.5). In addition to the two special purpose registers, the SoC design includes one or several chip unique secret key(s), stored in non-volatile registers. They are used to decrypt and check the integrity of security critical code/data that is loaded into the chip internal or external RAM. To prevent any usage of the chip unique secrets in normal mode, they are tied to the mode state register and locked to protected

[4]As discussed in Section F.5.2, unprivileged software can at most achieve a soft reboot to protected mode or a cold reboot.

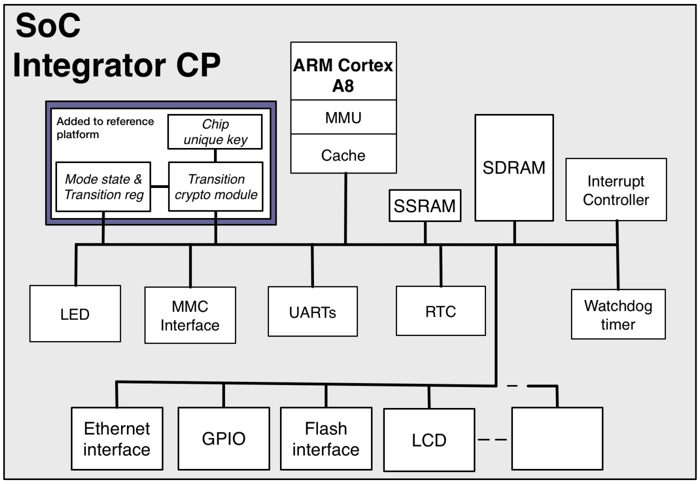

Figure F.2: SoC system view.

mode. In our proof of concept implementation we have optimized performance with a fully functional cryptographic module, the *transition crypto module*. However, cryptographic operations can be performed in software as well, reducing the number of changes to integrated circuits, but at the prize of an increased performance overhead. If not mentioned otherwise, we assume the presence of a transition crypto module in the remainder of the paper.

In order to show how these SoC components are used in the suggested approach, below we describe the details of the cold boot, the transition from protected to normal mode and the transition from normal to protected mode.

The flowchart in Figure F.3 summarizes the course in respect to the bootloader code.

Cold Boot The following steps are performed in a cold boot:

1. After the machine is powered on, a first stage boot code is executed. To prevent security from being compromised, this code needs to be protected from modifications by storing it in write-protected memory such as on-chip-ROM.

2. The first stage boot code loads the integrity protected second stage boot code and boot configurations into on-chip-RAM. The second stage boot code and its configurations are protected with signatures verifiable with a public key stored in write-protected memory, such as ROM, or hardware registers, such as e-fuse registers.

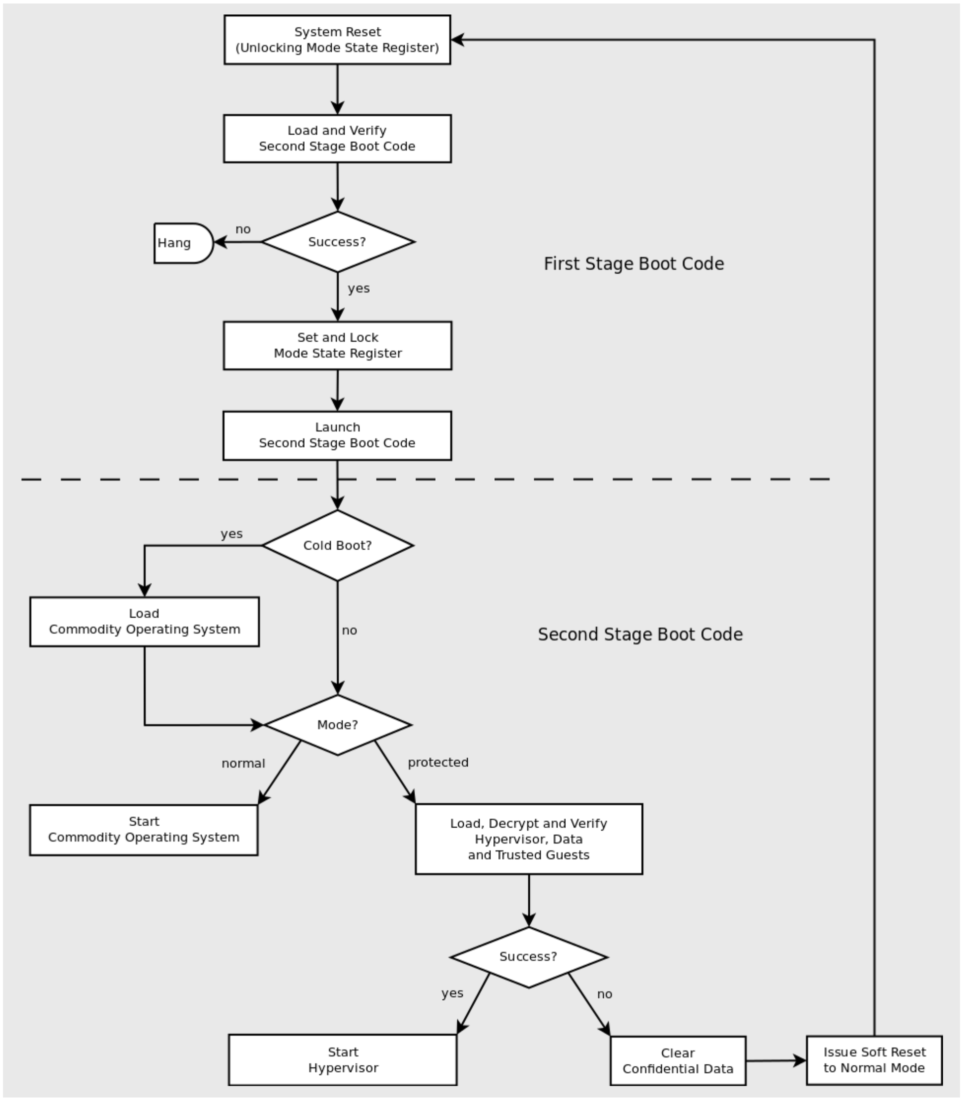

Figure F.3: Flowchart

3. The first stage boot code reads the verified boot configurations and writes the default boot mode (normal or protected) into the mode state register, which is then locked.
4. The first stage boot code launches the second stage boot code. Depending on the system and use case, one or several intermediate boot stages are processed until the boot code responsible for hypervisor or operating system launch is reached. We call this boot stage *transition boot stage*.
5. The transition boot stage reads the current value of the mode state register. If the register indicates normal mode, the operating system indicated in the boot configurations is launched. If the register indicates protected mode, the following steps are performed:
   a) The transition boot code loads hypervisor, trusted guest(s) and data from external memory and verifies the integrity (e.g., by using a transition crypto module). The confidentiality of trusted guests is protected through fast symmetric encryption with a chip unique secret key. If required, confidentiality protection can also be applied to the hypervisor or parts thereof.
   b) If decryption and integrity verification in the previous step were successful, the transition boot code hands over the execution to the hypervisor. Otherwise, the transition boot stage code clears all security sensitive data on the system, writes "normal mode" into the transition register and issues a soft reset, so that the system reboots into normal mode. This allows the system to recover even if it could not be started into protected mode.

Transition from Protected to Normal Mode When the system is in protected mode and secure services are no longer needed on the system, the hypervisor switches the system back to normal mode, as follows:

1. All trusted guests currently running are halted by the hypervisor. If required, persistent data is stored, integrity and confidentiality protected. Subsequently, the memory of trusted guests is cleared.
2. All confidential hypervisor data is cleared from memory.
3. The hypervisor can choose to maintain non-confidential code/data in memory to avoid reloading and reinitializing when returning to protected mode. In that case, *Message Authentication Codes* (MACs) protecting the integrity are recomputed, given that the concerned memory regions have changed.
4. The hypervisor sets the transition register to "normal" and issues a SoC-wide (soft) reset signal. This can be done via the component containing the two special purpose registers. The resulting soft reboot of the system will keep the

content of most volatile memories, which allows a rather quick booting process without the need to reload all code and data from non-volatile memories.

At reset, the system will be booted into normal mode (analogous to the previous paragraph) running the OS kernel in the most privileged CPU mode as “usual”, i.e., as in a non-virtualized system (see Figure F.1). Before handing over execution to the commodity OS, the boot code clears all registers to avoid that confidential data from a protected mode phase is leaked into normal mode.

Transition from Normal to Protected Mode When the system is in normal mode and one or several security critical services are required, the commodity operating system writes “protected” into the transition register and issues a soft reset signal. It can inform the hypervisor about requested services and their parameters by writing service request values into dedicated transition memory before the reset. Subsequently, the boot is performed in analogy to the cold boot into protected mode, retrieving mode information from the transition register. However, the commodity OS is not loaded again and, if chosen so, the non-confidential parts of the hypervisor (such as code, page tables, constants) are not either. In contrast to that, integrity verification is always performed, possibly even for new memory regions used by, for example, page tables created in the previous hypervisor session. If hypervisor memory has been compromised in normal mode or protected mode has not been active before, a fallback option will (re-)load the entire hypervisor from the storage as done in cold boot. Once the system is rebooted, the hypervisor will check the requested secure service(s) by reading the transition memory and launch them with the given parameters after checking that both services and parameters are valid and sound. Alternatively, this information can be passed via a hypercall from the commodity OS, once it is invoked by the hypervisor.

On a mode transition in either direction the commodity OS is usually aware of the upcoming soft reboot and will pause active processes as well as store their contexts before releasing control. Those processes (kept in memory) can then easily be resumed in the new mode. Before the hypervisor or OS reconfigures the peripherals, it needs to check whether interrupts (masked throughout the soft reboot) have occurred. Depending on the use case, the boot code can also be used to record events in a queue. In typical scenarios, the user will be aware of the inherent latency.

F.2.1 Implementation Alternatives

Enforced Protected Mode through Watch Dog Timer An alternative realization of the presented approach connects the watch dog timer of the SoC to the mode state register, so that the timer can *only* be reset if the system is in protected mode. If not kept alive, the watch dog issues a soft reset. At soft reset, the transition boot stage code checks the status of the watch dog timer and if it has reached zero, the transition boot code will boot the system into protected mode, *independently* of the transition register. This *forces* the system into protected mode in some pre-defined time intervals, which can be useful for monitoring or to counteract denial of other trusted services.

Soft Reboot Enabled by TrustZone The ARM TrustZone technology for ARM11 and ARM Cortex embedded processors [11] offers support for creating two securely isolated virtual cores (or *worlds* as they are termed) on a single real core. Both *secure world* and *normal world* manage an own virtual MMU, as well as an own vector table and thus own exception handlers [59]. System hardware, including memory and peripherals, can be allotted to each world. This is realized by an additional address bit. However, that separation requires that peripheral devices are adapted to the setting. A transition between the worlds is initiated by a hardware interrupt or a *Secure Mode Call* (SMC), both invoking the so called *monitor mode*, which is responsible for context switches. The concept of turning a hypervisor on and off on demand, as described in this paper, can also be implemented based on TrustZone instead of the discussed hardware extensions. Bootloader, hypervisor, trusted guest and the current mode would then be kept stored in the memory of the secure world, which only executes code to realize the soft reboot transitions. The execution of all other software (including the hypervisor and the trusted guest) happens in the normal world. Soft resets would be realized through SMCs. One of the advantages of this variant is that no soft reboot specific hardware extension in form of, e.g., a mode state register is required, something which is especially useful when TrustZone is already present anyway. Furthermore, keeping assets in the secure world reduces the need for crypto operations considerably. However, a secure boot scheme would still be needed to ensure that the hypervisor and the trusted guest(s) are loaded into the secure world memory confidentially and integrity protected. Hardware protected keys are therefore still required. Moreover, peripherals have to be adapted in order to maintain separation between the two worlds. This limitation together with the costs of the TrustZone extension makes a TrustZone driven implementation variant only preferable to the standard one if the soft reboot is to be enabled on an already existing system that (including its peripherals) supports TrustZone from the beginning.

Multicore Systems The presented solution is also applicable to multicore systems. Since the mode state is a global property to control access to the chip unique keys, all CPUs have to agree on the mode. Consequently, when in protected mode, all CPUs need to be protected by a hypervisor, irrespectively if they are running secure services or not. There would be some *master* hypervisor on the system, which has the responsibility to coordinate, to execute trusted services and to issue soft reboots. In order to switch from protected to normal mode, the master hypervisor would inform its neighbors and wait until it has received acknowledgments from all of them before issuing the actual reset signal. Likewise, when booting into protected mode, the master hypervisor will be booted first on the main CPU and then launch all other hypervisors.

F.3 Hypervisor

A prototype implementation for the described solution has been established on the basis of a type-1 hypervisor[5], available as open source from [162]. Its focus lies on providing security by MMU-supported separation and its isolation properties have been formally verified on binary level [53]. Following the system virtualization principle, it allows the parallel execution of multiple paravirtualized guests in user mode. Both Linux and FreeRTOS have been ported to the hypervisor. Isolation between guests can intentionally be relaxed by the possibility to communicate with well-defined and parameterizable RPCs via the hypervisor. In addition to inter-guest-separation, the hypervisor offers introspection features such as virtual guest modes that enable intra-guest-separation as well, for example in order to maintain the guest OS' kernel separation even when executing in the processor's non-privileged operation mode. The implementation of the hypervisor comprises 2717 lines of C code and 942 lines of assembly, resulting in a compiled binary of 31 KB. The hypervisor was developed for single-core ARMv5 and ARMv7 architectures and deployed on Beaglebone [31], Beagleboard [29], Beagleboard-xM [30], NovaThor [166] and the Integrator development board [15], as well as on emulated platforms within the OVP framework [133].

F.4 Software Adaptions

We have implemented a single-core prototype of the solution, based on FreeRTOS as commodity OS and the inhouse hypervisor for ARMv7 described in Section F.3. Both FreeRTOS and the hypervisor had to be modified to support the soft reboot functionality, as described in this section. The trusted domain was easily implemented since it only needs to offer an entry point for receiving RPCs and the awareness about the RPC parameter passing protocol. Three different interrupt vector tables were configured and are mapped according to the mode; while the vector of the boot code is only referred to on reset, the hypervisor vector is active in protected mode and FreeRTOS' vector is either referred to directly (in normal mode) or used to receive control from the hypervisor. Otherwise, memory mapping is static and access rights only change in dependency to the current mode. Binaries are linked/built separately for each entity and, where required, encrypted and/or integrity protected before deployment.

Adaptations to the Commodity OS The core adaptation in the commodity OS consists of changes that enables it to run both as guest on top of the hypervisor

[5]Hypervisors of type 1, also called native hypervisors, are not running on any host OS, but on bare metal.

and natively on bare metal with control over the privileged operation ring of the CPU. While in the latter setting, privileged operations are performed directly by the corresponding privileged instructions, hypercalls have to be used in the first setting. We added a dual API layer that selects the required implementation for each functionality in dependency of a mode indicating configuration bit set by the bootloader. Similarly, FreeRTOS was made able to switch between its own kernel separation enforcement and the kernel protection provided by the hypervisor. On startup, the commodity OS either performs its own hardware configurations or it registers itself to the hypervisor, before creating or resuming processes. Finally, we inserted code that makes use of the RPC functionality to communicate with a trusted domain and that actually initiates soft reboots for demonstration and benchmark purposes.

Adaptations to the Hypervisor The adaptations to the hypervisor were quite limited. Essentially, besides providing configuration information about commodity OS and trusted guest, only a hypercall needed to be added, that realizes the initiation of a soft reboot into normal mode, including the optional write back of the trusted domain and the erasure of all confidential data. The hypervisor makes use of the possibility to be partly kept in memory on soft reboot. In particular, this applies to the sections for code and constants, that both do not change throughout the system's uptime, and the page tables, for which a new MAC is computed after they are generated. Data section, BSS section, heap and stack are treated confidential and cleared before soft reboot. The data section is the only part that needs to be reloaded when coming back to protected mode, given that no memory corruptions have occurred in normal mode. Whether the hypervisor memory is still uncorrupted or had to be reloaded by the bootloader is indicated as argument to the hypervisor, so that page tables can be recomputed if necessary. Note that we migrated the responsibility of loading guests from the hypervisor to the bootloader. Similarly, we decided to invoke trusted services via RPCs by the commodity OS after a soft reboot to protected mode instead of passing parameters about the desired service to the hypervisor. In that way, no decisions are required by the hypervisor upon boot, but control can simply be transfered to the guest's entry point directly.

Bootloader The implementation of the bootloader was carried out in a straight forward manner according to the protocol in Section F.2, using a transition crypto module for cryptographic operations. In our implementation the bootloader is divided into two stages. The first stage boot code checks the transition register, loads and verifies the second stage boot code and is placed into ROM along with its vector table. The second stage boot code loads the commodity OS, the hypervisor and the trusted guest (depending on the mode), carries out needed verification steps and finally calls the commodity OS or hypervisor.

F.5 Evaluation

The approach can be implemented on many current embedded architectures with minor hardware changes (a few special purpose registers and hardware protected keys), as most of the functionality relies on existing hardware features and functions implemented in software, mainly the boot code and the hypervisor. To demonstrate our solution and in order to obtain benchmarks on its performance, we have implemented the described hardware extensions within the emulation framework OVP [133]. It allows to implement and simulate the behavior of new SoC hardware components with reasonable effort. The additional registers are realized as memory mapped device connected to a SoC (emulation) with an integratorCP platform that includes a single ARM Cortex-A8 CPU. The register extension has been wired to perform system resets when required. Furthermore, a dedicated transition crypto module has been modelled in OVP, allowing us to verify the required encryption/decryption and integrity check tasks. As OVP can not provide exact simulation times, especially with respect to peripherals, MMU and caches, the main purpose has been to test the concept as such and get a good picture of the performance one can expect. Hence, the transition crypto module is simplified with respect to its hardware interface and we allow direct memory read from the transition crypto module over the bus. This allows us to test the different boot cases and the concept, but not to simulate real transition crypto module data transfers from the CPU or via DMA. We believe those simplifications are reasonable since exact time estimates for these access forms can not be obtained in an emulation environment such as OVP anyway.

F.5.1 Performance

The suggested approach allows running secure services isolated by a hypervisor layer only when needed instead of permanently. Consequently, the secure services can be implemented with a very small performance impact. This comes at the price of soft resets when the secure services are needed. The objective of our benchmarks is thus to estimate the overall costs for a soft reboot.

The evaluation includes three factors:

- the number of bytes copied (or erased) between/from storage devices (NAND flash, RAM),
- the number of bytes fed into the transition crypto module for en-/decryption or integrity value calculation and integrity checks,
- the number of remaining CPU instructions not involved in any such feeding, copying or clearing operations.

These figures together allow us to estimate the overall time for all steps of the suggested approach, making the following assumptions about the platform:

- The CPU is clocked with 720 MHz and nominally executes 200 MIPS, as typical on many Cortex A8 development boards such as BeagleBoard [29].
- We assume a rather conservative RAM copy speed of 150 MB/second, which is a lower estimate from [17].
- The copy speed from NAND flash to RAM is estimated by 6 MB/second [114].
- We assume a transition crypto module supporting SHA-256 HMAC generation and AES-128 en-/decryption with a fair trade off between size and speed, clocked at 174 MHz and with the ability to perform parallel hashing/encryption or hashing/decryption with the speed of 171 MB/second. Since hashing is the dominating work load in such parallelized operations, the feasibility of such a speed can be concluded from, e.g., [45].

Table F.1 provides an overview of the results for the single steps required, depending on which transition is been considered. We distinguish between a cold boot into protected mode (cp), a cold boot into normal mode (cn), a (warm/soft) reboot into protected mode (wp) and a (warm/soft) reboot into normal mode (wn). Crosses (X) indicate which step is involved in which transition. A dash (-) indicates that the step in question is optional or does only occur in the first of typically many soft reboots.

The benchmarks are based on a second stage boot code of 2.9 KB, FreeRTOS as commodity operating system with a binary blob of 1 MB, the hypervisor sections for code and constants, together 30 KB, hypervisor data of 1 KB and a trusted domain of 380 KB. Those specifications refer to the initial volumes. However, we allow the trusted domain to grow up to 1 MB for the usage of stacks, data structures etc. The space reserved for the hypervisor's heap, stack and BSS section is 900 KB, while page table memory can be up to 64 KB.

A complete soft reboot cycle including two mode switches is with 19 milliseconds estimated considerably faster than any cold boot, irrespectively of the targeted mode. Avoiding slow accesses to external storage is responsible for the main share of those performance benefits. However, also the number of boot instructions is reduced in warm reboot, in respect to both the hypervisor and the commodity OS. In both cases this optimization is mainly due to the dispensed page table reconfiguration. Preconfigured page tables could reduce the hypervisor's booting phase also in a cold boot, but that would come at the costs of an increased foot print and less flexibility. Since we allow the trusted guest to grow to a size of up to 1 MB, writing it back (including MAC computation and encryption) is comparatively expensive, and so is its deletion. In order to optimize write back and clearing, one would need to narrow down the space actually claimed by the trusted guest. However, writing the trusted guest back might not be needed in many cases and is therefore listed as optional. The share of cryptographic operations on the estimated costs of a warm reboot to protected mode is 88%. Clearing confidential data is constituting the main part of the costs when soft rebooting into normal mode. We believe that the soft reboot performance is more than reasonable in settings where a hypervisor is only needed sporadicly. Assuming the estimations from above and a hypervisor overhead of at least 2%, an execution phase of 1 second in which secure services are not required is already enough to make a temporarily deactivation of service and hypervisor through a soft reboot profitable. As the soft reset, different from a full reset, keeps all volatile memory content, soft reboots are also considerably faster than cold reboots with full resets. In order to achieve the same functionality of enabling and disabling virtualization on demand with full resets, additional costs to the ones listed above would arise, for example for storing application data before rebooting.

F.5.2 Security

F.5.2.1 Attacker Model

We assume that the attacker has full control over the commodity OS. However, the hypervisor is supposed to be free from vulnerabilities, which can be assured by formal verification. Furthermore, we trust CPU, MMU and BIOS. Hardware attacks are out of scope of this paper. Devices are assumed to reset whenever a reset signal is issued.[6] In particular, no previously pending DMA operations will be performed after the reset until DMA controllers are reprogrammed. We furthermore assume that the hypervisor is aware of the specification of all present DMA devices, so it can intercept accordingly, and that the devices' behavior actually follows their (non-hostile) specifications. Alternatively, an IOMMU can be used to protect against DMA attacks.

We assume that the attacker aims at obtaining confidential data about the trusted guest and/or to affect its execution outside of the controlled communication channel provided. Denial of Service (DoS) attacks are out of scope of this paper, since a malicious commodity OS has the ability of shutting down the machine or otherwise introducing delays anyway. However, making the watch dog timer aware of the mode state register as described in Section F.2.1 improves the protection against DoS attacks, even though complete protection is not achieved by this enhancement either.

F.5.2.2 Protection in Different Execution Phases

In the following, we discuss the different aspects of the system's security in detail.

[6]For functionality, the operating system or hypervisor respectively needs to wait until devices have finished pending tasks before issuing a reset signal. However, the specific time of a reset has no effect on the security.

Execution in Normal Mode When in normal mode, the trusted guest and confidential parts of the hypervisor are stored in encrypted form. Access to the corresponding chip unique key(s) is rejected.

Entering Protected Mode The system can only enter protected mode along with the execution of a trusted and unmodifiable bootcode. In order to change the mode register, it needs to be unlocked. It is guaranteed by hardware that this unlocking is performed together with a CPU reset. The reset sets the program counter to a fixed address pointing to the bootcode in ROM. On ARM processors, neither this address nor the endianess or the instruction set used after reset can be changed by the commodity OS, even when running in privileged mode, since the values for those system parameters are copied from the System Control Register (`SCTLR`) register of coprocessor 15 which in turn is set back to default values first on reset [13, pp. B1-1202, B1-1203]. In particular, the MMU is disabled [13, p. B3-1308], so that the used entrance point of the exception vector table can not be translated to a different address. Standard interrupts are masked by the reset and not unmasked before control has been transferred to the hypervisor. Fast interrupts are disabled by the boot code, even though there are no devices tied to fast interrupts in our setting. The remaining bits of the Current Processor State Register (`CPSR`) are set to default values by the boot code. If the integrity verification of either hypervisor or trusted guest fails, the memory is cleared and a reset to normal mode is enforced, so that compromised software will never be executed.

Execution in Protected Mode The hypervisor is the first software invoked by the boot code. It configures the system's memory protection in such a way that the hypervisor code and data, the trusted domain, the transition crypto module, chip unique keys and register extensions are inaccessible to guests. All exceptions are mapped to handlers under the control of the hypervisor.

Leaving Protected Mode In order for the commodity OS to (re-)gain privileged rights, a reset has to be issued, since the hypervisor is maintaining control over the system in all other cases. From a functional perspective, this is ideally done through the hypervisor by sending an unlock request to the mode register. However, from a security point of view we have to assume that the attacker can establish a reset signal at any arbitrary time. In case this happens when the transition register is (still) set to protected, the system will either get back to a state where the hypervisor is in charge or (if integrity verification fails) all data will be erased and the mode changed to normal before booting the commodity OS. Even achieving one or several more reset signals during the soft reboot process will not be of any benefit to the attacker since she has no possibility to set the transition register to normal during that phase. In the other case that the transition register is set to normal before reset, the system has either been in normal mode anyway (and confidential data is not present) or the hypervisor has already erased all confidential data (as required by the protocol before setting the transition register back to normal). The MMU is preventing unprivileged access to the transition register. Multiple randomized overwriting of confidential memory regions can be used instead of single overwriting, if deleted information must to not be retrievable in hardware forensics. Before handing over execution to the commodity OS, the boot code clears all registers to avoid that confidential data from a protected mode phase is leaked to normal mode.

F.5.2.3 Further Aspects

DMA Devices In normal mode, devices do not have any more privileges than the commodity OS. In protected mode, the hypervisor is able to intercept all attempts to program DMA devices or can configure an IOMMU to protect security critical parts of the memory. On soft reboot, pending DMA tasks are canceled. In particular, the only DMA operations performed during the booting phase are those executed with respect to the (trusted) transition crypto module.

Proof of Mode A design assumption of our solution was that the fact that the system is running in protected mode will be proven to the user by functionality. For many common applications (e.g., for secret key services such as signing) it is impossible for the attacker to make the user believe the trusted application was active if it was actually not. However, alternative embodiments are possible where a secret is displayed to the user or a LED is tied to the mode register.

F.6 Related Work

In [85] IBM describes a method for directing the system's reset signal to a specific partition in a virtualized setting. The method is therefore another suggestion on how to make use of reset functionality in virtualized environments, but does not address virtualization overhead.

Instead of disabling virtualization completely when it is not needed, a natural first step is to reduce its costs to a minimum. For example, in specific I/O operations hypervisors can be bypassed [113]. However, this requires hardware support and applies only to a subset of all (I/O) operations. Naughton et al. [128] discuss approaches to extend the Xen hypervisor dynamically by loading additional modules on runtime. In that way, the usage of space and other resources can be optimized. Still, a basic instance of Xen would always be active, something we avoid in our solution.

How to turn off a hypervisor while keeping other software running has been demonstrated for a machine with a dedicated processor mode for virtualization [73]. However, in many embedded architectures - for example on the common ARMv7-processor - the additional requirement of lifting the operating system to the privileged ring needs to be accomplished as well. Furthermore, the soft reboot approach described in the present paper allows turning on the hypervisor (again), guaranteeing the integrity protection of both the booted hypervisor and additional guests while the hypervisor is off.

The separation facilities provided by TrustZone (see Section F.2.1) can be used to execute trusted services isolated without suffering from the performance overhead introduced by virtualization and without the need of paravirtualizing the commodity OS. At the same time, other CPUs on the system stay unaffected, which can be seen as additional advantage over the soft reboot approach, which requires all CPUs to agree on the mode. However, even if considering a system with a CPU already supporting TrustZone (which is not given for many embedded processors, such as CPUs with ARMv5 architecture), using TrustZone to execute software isolated requires from the SoC that peripherals are adapted in order to respect the extended address format and thus maintain separation between the two worlds. In contrast, the solution presented in this paper requires only minor additions to the SoC. If the execution of several isolated services or a symmetric protection between service(s) and commodity OS is required in a TrustZone solution, the secure world will need to run a separation kernel, as used in the proposed soft reboot solution as well. Note that TrustZone based approaches still need to make sure that trusted services are kept confidential before being loaded from external storage to the secure world. To achieve this, further hardware extensions are required in order to provide a secure boot scheme.

An alternative way to securely invoke a hypervisor at an arbitrary point of time is provided by *trusted computing* technology [174]. Similar to our solution, trusted guests (and hypervisor) would be kept encrypted and integrity protected until a cryptographic hardware module (in that case the *Trusted Platform Module* (TPM)) decrypts and verifies them. However, in this method called *sealed storage*, the collaboration of the decrypting module does not depend on a mode, but on binaries loaded to the system. Applying the *late launch* technology, as available for modern Intel and AMD processors, this check ignores already loaded software and instead ensures that a dedicated secure load block (SLB) is executed. Only a loaded and unmodifed SLB will enable the decryption of the sealed data [82, 1, 116]. This principle is comparable to the entanglement of the mode register's unlocking and the reset that enforces the execution of the first stage boot code in our approach. However, not only is the technology not available for embedded systems, it has also been demonstrated that late launch can be circumvented and hypervisors can be modified by malicious code injected to the system before the late launch [182, 153]. Even if this attack cannot be applied to all architectures and the vulnerability might be fixed in the future, it gives reason to doubt that TPM-based solutions provide a holistic principle covering the entire system. Furthermore, TPM-operations are comparatively expensive, due to a slow bus connection and relatively slow asymmetric decryption algorithms. A proper (and still simple) mode aware cryptographic module (with DMA support), which we suggest for our approach, is more efficient and cost-effective and does not require any modifications to the CPU.

Making use of the same enablers (sealed storage and late launch), the Flicker environment [116] focuses on the isolated execution of single trusted applications instead of the delayed activation of a hypervisor. This decision against virtualization certainly decreases the trusted computing base even more, but comes with the drawback that the commodity operating system has to be paused while the trusted application is being executed and that only one trusted service can be active at a time. A similar functionality to the one of Flicker can be achieved with the hardware extensions that we propose. However, the feature of remote attestation is naturally reserved to platforms with trusted computing support. Furthermore, [116] admittedly provides a stronger protection against replay attacks even without further hardware extension.

SICE [21] makes use of x86's System Management Mode (SMM) to provide an asymmetric isolation between commodity OS and isolated software, based on a TCB including only the hardware, the BIOS and the SMM with a software foundation of 300 LoC (excluding cryptographic libraries). However, isolated software can not access peripherals directly and - as the authors point out themselves - since the SMM was not designed with security in mind and several attacks on it are already known, careful security reviews are necessary before deployment. While still seeming to be a promising approach for asymmetric isolation on x86 systems, SICE's principle is not applicable to embedded systems.

F.7 Conclusion

We have presented a dual mode approach to turn the system's hypervisor on and off on demand. Integrity and privacy of trusted guests are maintained at all times: while virtualization is active (in *protected mode*), while it is not (in *normal mode*), and while the machine is powered off. The solution requires only minor additions to an existing SoC design, namely two new registers and hardware protected keys. Hardware support for the cryptographic operations guarantees efficiency. No extensions to the CPU or adaption of other devices are needed. The performance measurements of a prototype implementation in emulated hardware show that soft reboots can provide benefits in several scenarios for embedded systems. In particular, the efficiency is higher than when performing a cold reboot or maintaining virtualization while not needed. The main costs for enabling isolated services consists of their decryption and the integrity check of those services and of the lightweight hypervisor. Returning to non-virtualized execution does not take much longer than the erasure of newly produced confidential data. Furthermore, paravirtualization is not necessary in settings where the commodity OS can be paused while in protected mode. We leave the formal verification of our approach as possible future work.

step	cp	cn	wp	wn	Bytes accessed in storage	crypto module load in B	other instr.	estimated time in ms
configure registers and mode	X	X	X	X			29	0.0001
clean registers				X			41	0.0002
load + verify 2nd stage code	X	X	X	X	2,987	2,987	26	0.4916
load FreeRTOS	X	X			1,043,288		19	165.8263
load + verify hypervisor code	X				30,500	30,500	24	5.0181
verify hypervisor code			X			30,500	22	0.1940
load + verify hypervisor data	X		X		960	960	22	0.1581
verify hypervisor page tables			X			65,536	21	0.4168
copy encrypted trusted guest	X				389,732		0	61.9562
decrypt + verify trusted guest	X		X			389,732	24	4.9558
boot hypervisor	X						247,940	1.2397
*re*boot hypervisor			X				27,707	0.1385
boot FreeRTOS, normal mode		X					9,146	0.0457
*re*boot FreeRTOS, normal mode				X			140	0.0007
(re-)boot FreeRTOS, protected	X		X				305	0.0015
compute page table MAC				-		65,536	129	0.4173
(write back trusted guest)				-		1,048,576	216	13.3341
erase confidential memory				X	1,964,252		98	12.4889
initiate reset to protected mode			X				41	0.0002
cold boot, protected mode	X						248,389	239.6374
cold boot, normal mode		X					9,220	166.3637
warm reboot, protected mode			X				28,197	6.3566
warm reboot, normal mode				X			334	12.9815

Table F.1: Execution Costs per Step

Reference Lists

SICS Dissertation Series

Dissertation series of SICS Swedish ICT:

1. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.
2. Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.
3. Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.
4. Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.
5. Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.
6. Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.
7. Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.
8. Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.
9. Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.
10. Mats Björkman, Architectures for High Performance Communication, 1993.
11. Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.
12. Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.
13. Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.
14. Sverker Jansson, AKL – A Multiparadigm Programming Language, 1994.
15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.
16. Torbjörn Keisu, Tree Constraints, 1994.
17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.
18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.
19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.
20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.
21. Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.
22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.

23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.
24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.
25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.
26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.
27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.
28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.
29. Lars-Åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.
30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.
31. Fredrik Espinoza, Individual Service Provisioning, 2003.
32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.
33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.
34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.
35. Emmanuel Frécon, DIVE on the Internet, 2004.
36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.
37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.
38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.
39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.
40. Markus Bylund, A Design Rationale for Pervasive Computing – User Experience, Contextual Change, and Technical Requirements, 2005.
41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.
42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.
43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.
44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.
45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.
46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007
47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007
48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008
49. Daniel Gillblad, On Practical Machine Learning and Data Analysis, 2008

50. Fredrik Olsson, Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora, 2008
51. Ian Marsh, Quality Aspects of Internet Telephony, 2009
52. Markus Bohlin, A Study of Combinatorial Optimization Problems in Industrial Computer Systems, 2009
53. Petra Sundström, Designing Affective Loop Experiences, 2010
54. Anders Gunnar, Aspects of Proactive Traffic Engineering in IP Networks, 2011
55. Preben Hansen, Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships, 2011
56. Fredrik Österlind, Improving Low-Power Wireless Protocols with Timing-Accurate Simulation, 2011
57. Ahmad Al-Shishtawy, Self-Management for Large-Scale Distributed Systems, 2012
58. Henrik Abrahamsson, Network overload avoidance by traffic engineering and content caching, 2012
59. Mattias Rost, Mobility is the Message: Experiment with Mobile Media Sharing, 2013
60. Amir H. Payberah, Live Streaming in P2P and Hybrid P2P-Cloud Environments for the open Internet, 2013
61. Oscar Täckström, Predicting Linguistic Structure with Incomplete and Cross-Lingual Supervision, 2013
62. Cosmin Arad, Programming Model and Protocols for Reconfigurable Distributed Systems, 2013
63. Tallat M. Shafaat, Partition Tolerance and Data Consistency in Structured Overlay Networks, 2013
64. Shahid Raza, Lightweight Security Solutions for the Internet of Things, 2013
65. Mattias Jacobsson, Tinkering with Interactive Materials: Studies, Concepts and Prototypes, 2013
66. Baki Cakici, The Informed Gaze: On the Implications of ICT-Based Surveillance, 2013
67. John Ardelius, On the Performance Analysis of Large Scale, Dynamic, Distributed and Parallel Systems, 2013
68. Fatemeh Rahimian, Gossip-Based Algorithms for Information Dissemination and Graph Clustering, 2014
69. Rebecca Steinert, Probabilistic Fault Management in Networked Systems, 2014
70. Mudassar Alsam, Bringing Visibility in the Clouds: Using Security, Transparency and Assurance Services, 2014
71. Anna Ståhl, Designing for Interactional Empowerment, 2015
72. Pedro Sanches, Health Data: Representation and (In)visibility, 2015
73. Tomas Olsson, A Data-Driven Approach to Remote Fault Diagnosis of Heavy-duty Machines, 2015
74. Nicolas Tsiftes, Storage-Centric System Architectures for Networked, Resource-Constrained Devices, 2016
75. Oliver Schwarz, No Hypervisor Is an Island: System-wide Isolation Guarantees for Low Level Code, 2016

Bibliography

[1] Advanced Micro Devices. *AMD64 virtualization: Secure virtualization: Secure virtual machine architecture reference manual*, 2005. Publication number 33047, revision 3.01.

[2] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*, November 2009. Publication number 34434, revision 1.26.

[3] Eyad Alkassar, Ernie Cohen, Mikhail Kovalev, and Wolfgang J. Paul. Verification of TLB virtualization implemented in C. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments: 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, pages 209–224, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-27705-4. URL `http://dx.doi.org/10.1007/978-3-642-27705-4_17`.

[4] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In *Verified Software: Theories, Tools, Experiments*, 2008.

[5] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proc. VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2010.

[6] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel. In *Verified Software: Theories, Tools, Experiments*, 2010.

[7] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22791-5. URL `http://dx.doi.org/10.1007/978-3-540-27864-1_10`.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent R. 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, HASP '13, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1.

[9] Oana Fabiana Andreescu, Thomas Jensen, and Stéphane Lescuyer. Correlating structured inputs and outputs in functional specifications. In Rocco De Nicola and Eva Kühn, editors, *Proceedings of Software Engineering and Formal Methods: 14th International Conference, SEFM 2016*, pages 85–103. Springer International Publishing, 2016. ISBN 978-3-319-41591-8.

[10] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy*, page 0065, 1997. ISSN 1540-7993.

[11] ARM. ARM TrustZone technology. `http://www.arm.com/products/processors/technologies/trustzone.php`.

[12] ARM. ARMv7-A architecture reference manual, issue B.

[13] ARM. ARMv7-A architecture reference manual, issue C. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c`.

[14] ARM. Cortex-A15 processor. `http://www.arm.com/products/processors/cortex-a/cortex-a15.php`.

[15] ARM. Integrator baseboards. `http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.integratorbaseboards`.

[16] ARM. *PrimeCell DMAController (PL080) Technical Reference Manual*, rev r1p3 edition, 2005. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0196g/index.html`.

[17] ARM Technical Support Knowledge Articles. What is the fastest way to copy memory on a Cortex-A8? `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html`, 2011.

[18] François Armand and Michel Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In *Proceedings of the 6th Consumer Communications and Networking Conference (IEEE CCNC '09)*, Las Vegas, NV, USA, January 2009.

[19] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8. URL `http://dx.doi.org/10.1007/978-3-540-88313-5_22`.

[20] Thomas H. Austin, Cormac Flanagan, and Martín Abadi. A functional view of imperative information flow. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35181-5. URL `http://dx.doi.org/10.1007/978-3-642-35182-2_4`.

[21] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[22] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Principles of Programming Languages*, POPL, pages 165–178. ACM, 2014. ISBN 978-1-4503-2544-8. URL `http://doi.acm.org/10.1145/2535838.2535839`.

[23] Musard Balliu, Mads Dam, and Roberto Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 1080–1091. ACM, 2014. ISBN 978-1-4503-2957-6. URL `http://doi.acm.org/10.1145/2660267.2660322`.

[24] 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. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 164–177, New York, October 19–22 2003. ACM Press.

[25] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In Michael Butler and Wolfram Schulte, editors, *Proc. FM'11*, volume 6664 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2011. ISBN 978-3-642-21436-3.

[26] 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`.

[27] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security (SAFECOMP)*, pages 187–200. Springer, 2009. ISBN 978-3-642-04468-7.

[28] Max Bazaliy, Seth Hardy, Michael Flossman, Kristy Edwards, Andrew Blaich, and Mike Murray. Technical analysis of Pegasus spyware: An investigation into highly sophisticated espionage software. Whitepaper, Lookout, August 2016. `https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf`.

[29] BeagleBoard.org Foundation. BeagleBoard product page. `http://beagleboard.org/Products/BeagleBoard`.

[30] BeagleBoard.org Foundation. BeagleBoard-xM product page. `http://beagleboard.org/Products/BeagleBoard-xM`.

[31] BeagleBoard.org Foundation. BeagleBone product page. `http://beagleboard.org/Products/BeagleBone`.

[32] Michael Becher, Maximillian Dornseif, and Christian N Klein. FireWire: all your memory are belong to us. *Presentation, CanSecWest/core05*, 2005.

[33] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development : Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, Berlin, 2004. ISBN 3-540-20854-2.

[34] William R. Bevier. A verified operating system kernel. Technical Report 11, Computational Logic, Inc., Austin, Texas, USA, October 1987. `http://www.cs.utexas.edu/users/boyer/ftp/cli-reports/011.pdf`.

[35] William R. Bevier. Kit: a study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, Nov 1989. ISSN 0098-5589.

[36] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together – formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4): 411–430, 2006. ISSN 1433-2779. URL `http://dx.doi.org/10.1007/s10009-006-0204-6`.

[37] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[38] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

[39] Rolf Blom and Oliver Schwarz. High assurance security products on COTS platforms. *ERCIM News*, Special Theme: Trustworthy Systems of Systems (number 102):39–40, 2015. ISSN 0926-4981.

[40] Pauline Bolignano, Thomas Jensen, and Vincent Siles. Modeling and abstraction of memory management in a hypervisor. In *International Conference on Fundamental Approaches to Software Engineering*, pages 214–230. Springer Berlin Heidelberg, 2016.

[41] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press international series in formal methods. Academic Press Incorporated, 1988. ISBN 9780121229528.

[42] Jörg Brakensiek, Axel Dröge, Martin Botteck, Hermann Härtig, and Adam Lackorzynski. Virtualization as an enabler for security in mobile devices. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 17–22. ACM, 2008.

[43] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer, 2011. ISBN 978-3-642-22109-5.

[44] Reto Buerki and Adrian-Ken Rueegsegger. Muen - an x86/64 separation kernel for high assurance. Technical report, University of Applied Sciences Rapperswil (HSR), Switzerland, 2013. `https://muen.codelabs.ch/muen-report.pdf`.

[45] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. Improving SHA-2 hardware implementations. In *In Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006*, 2006.

[46] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 431–447. ACM, 2016. ISBN 978-1-4503-4261-2. URL `http://doi.acm.org/10.1145/2908080.2908101`.

[47] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *conf. theorem proving in high order logics (TPHOLS), volume 5674 of LNCS*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. ISBN 978-3-642-03358-2. URL `http://dx.doi.org/10.1007/978-3-642-03359-9`.

[48] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. URL `http://doi.acm.org/10.1145/800157.805047`.

[49] 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. ACM, 2016. ISBN 978-1-4503-4261-2. URL `http://doi.acm.org/10.1145/2908080.2908100`.

[50] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *LNCS*, pages 21–30. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25435-5. URL `http://dx.doi.org/10.1007/978-3-540-31987-0_3`.

[51] David Cyrluk, S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222. Springer, 1994. ISBN 3-540-59047-1. URL `http://dl.acm.org/citation.cfm?id=645903.672930`.

[52] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM virtualization: Performance and architectural implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, June 2016.

[53] 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 and Communications Security*, CCS, pages 223–234, 2013.

[54] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In *International Workshop on Trustworthy Embedded Devices (TrustED)*, 2013.

[55] Matthias Daum, Nelson Billing, and Gerwin Klein. Concerned with the unprivileged: user programs in kernel refinement. *Formal Aspects of Computing*, 26(6):1205–1229, 2014. ISSN 1433-299X.

[56] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. ISBN 978-3-540-78800-3.

[57] 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`.

[58] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. ARMvisor: System virtualization for ARM. In *Linux Symposium*, 2012.

[59] Heradon Douglas and Christian Gehrmann. Secure virtualization and multicore platforms state-of-the-art report. Technical Report T2009:14A, SICS Swedish ICT, Kista, Sweden, December 2009. `http://soda.swedish-ict.se/3800/`.

[60] Jianjun Duan. *Formal verification of device drivers in embedded systems.* PhD thesis, University of Utah, 2013.

[61] Jianjun Duan and John Regehr. Correctness proofs for device drivers in embedded systems. In *Proceedings of the 5th international conference on Systems software verification*, SSV'10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1929004.1929009`.

[62] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU system management mode to circumvent operating system security functions. In *Proc. CanSecWest*, 2006.

[63] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. *IEEE Security and Privacy*, 12(4):29–37, 2014. ISSN 1540-7993.

[64] Shawn Embleton, Sherri Sparks, and Cliff C. Zou. SMM rootkits: a new breed of OS independent malware. In *Security and Privacy in Communication Networks (SecureComm)*, pages 11:1 – 11:12, 2008.

[65] E. Allen Emerson. The beginning of model checking: A personal perspective. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking: History, Achievements, Perspectives*, pages 27–45. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69850-0. URL `http://dx.doi.org/10.1007/978-3-540-69850-0_2`.

[66] Richard J Feiertag and Peter G Neumann. The foundations of a provably secure operating system (PSOS). In *National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[67] Anthony C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2003.

[68] Anthony C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, pages 187–202, 2015.

[69] Anthony C. J. Fox, Michael J. C. Gordon, and Magnus O. Myreen. Specification and verification of ARM hardware and software. In S. David Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 221–247. Springer US, 2010. ISBN 978-1-4419-1539-9.

[70] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010. ISBN 978-3-642-14051-8.

[71] Jason Franklin, Arvind Seshadri, Ning Qu, Anupam Datta, and Sagar Chaki. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report CMU-Cylab-08-008, Carnegie Mellon University/Cylab, June 2008.

[72] FreeRTOS/real time engineers ltd. `http://www.freertos.org`.

[73] František Gábriš. Turning off hypervisor and resuming OS in 100 instructions. Presentation at FASM CON 2009, Myjava, Slovak Republic, `http://fdbg.x86asm.net/Turning_off_hypervisor_and_resuming_OS_in_100_instructions.ppt`.

[74] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007. ISBN 978-3-540-73367-6.

[75] Christian Gehrmann, Heradon Douglas, and Dennis Kengo Nilsson. Are there good reasons for protecting mobile phones with hypervisors? In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 906–911, January 2011.

[76] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. Physical key extraction attacks on PCs. *Communications of the ACM*, 59(6):70–79, 2016. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/2851486`.

[77] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[78] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86, April 1984.

[79] Robert P. Goldberg. *Architectural principles for Virtual Computer Systems.* PhD thesis, Harvard University, 1973.

[80] Robert P. Goldberg. Survey of virtual machine research. *IEEE Comp. Magazine*, 1974.

[81] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993. ISBN 0-521-44189-7.

[82] David Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing.* Books by engineers, for engineers. Intel Press, 2006. ISBN 9780976483267.

[83] 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.

[84] Jonas Haglund. Formal verification of systems software: No execution of malicious software in linux in networked embedded systems. master thesis at KTH Royal Institute of Technology, 2016.

[85] Bradley Ryan Harrington, Chetan Mehta, Devon Miller II Milton, Michael Anthony Perez, David Lee Randall, and David R. Willoughby. System and method for selectively executing a reboot request after a reset to power on state for a particular partition in a logically partitioned system. US patent US 7146515 B2, `http://www.google.com/patents/US7146515`.

[86] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for mikrokernel-based operating systems. Technical Report TUD-FI03-08, Dresden University of Technology, Dresden, Germany, 2003. `http://os.inf.tu-dresden.de/papers_ps/tr-ioarch-2003.pdf`.

[87] HASPOC project. `http://haspoc.sics.se/`.

[88] Steve Heath. *Embedded Systems Design.* Elsevier Science, second edition, 2002. ISBN 9780080477565.

[89] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Principles of Programming Languages*, POPL, pages 365–377. ACM, 1998. ISBN 0-89791-979-3. URL `http://doi.acm.org/10.1145/268946.268976`.

[90] 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`.

[91] 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`.

[92] M. A. Hillebrand, T. In der Rieden, and Wolfgang J. Paul. Dealing with I/O devices in the context of pervasive system verification. In *International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, pages 309–316, 2005.

[93] Charles Antony Richard 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`.

[94] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1. URL `http://doi.acm.org/10.1145/2487726.2488370`.

[95] HOL4 project. `https://hol-theorem-prover.org/`.

[96] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Principles of Programming Languages*, POPL, pages 79–90. ACM, 2006. ISBN 1-59593-027-2. URL `http://doi.acm.org/10.1145/1111037.1111045`.

[97] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004. ISBN 9781139453059.

[98] 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 *5th IEEE Consumer Communications and Networking Conference (CCNC 2008)*, Las Vegas, NV, USA, January 2008.

[99] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Technical report, IACR Cryptology ePrint Archive, 2015.

[100] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT) - Software Development Guide*, revision 012 edition, July 2015. `http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf`.

[101] Isabelle website. `https://isabelle.in.tum.de/`.

[102] Narges Khakpour, Oliver Schwarz, and Mads Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs (CPP)*, pages 276–291, 2013.

[103] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1): 27–69, February 2009. ISSN 0973-7677. URL `http://dx.doi.org/10.1007/s12046-009-0002-4`.

[104] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014. ISSN 0734-2071. URL `http://doi.acm.org/10.1145/2560537`.

[105] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 2009. ACM. ISBN 978-1-60558-752-3.

[106] Paul Carl 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`.

[107] Paul Carl Kocher. Computer security is broken: Can better hardware help fix it? *Commun. ACM*, 59(8):22–25, 2016. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/2955112`.

[108] Xeno Kovah and Corey Kallenberg. How many million BIOSes would you like to infect? *Whitepaper*, June 2015.

[109] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992. ISSN 0734-2071. URL `http://doi.acm.org/10.1145/138873.138874`.

[110] Donald C Latham. Department of defense trusted computer system evaluation criteria. Technical Report DoD 5200.28-STD, U.S. Department of Defense, december 1986.

[111] 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`.

[112] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Programming Language Design and*

Implementation, PLDI, pages 109–120. ACM, 2011. ISBN 978-1-4503-0663-8. URL `http://doi.acm.org/10.1145/1993498.1993512`.

[113] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[114] Make Linux Software. Super fast boot of embedded Linux. `http://www.makelinux.com/emb/fastboot/omap`.

[115] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[116] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42:315–328, 2008.

[117] 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`.

[118] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15, 2016.

[119] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13. ACM, 2013. ISBN 978-1-4503-2118-1. URL `http://doi.acm.org/10.1145/2487726.2488368`.

[120] Roberto Mijat and Andy Nightingale. Virtualization is coming to a platform near you. *ARM Whitepaper*, 2011. `http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`.

[121] Manoranjan Mohanty, Viktor Do, and Christian Gehrmann. Media data protection during execution on mobile platforms – a review. Technical Report T2014:02, SICS Swedish ICT, Kista, Sweden, July 2014. `http://soda.swedishict.se/5685/`.

[122] David Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Embedded Software*, 2007.

[123] μClinux. `http://www.uclinux.org/`.

[124] 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. IEEE Computer Society, 2013. ISBN 978-1-4673-6166-8.

[125] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2012. ISBN 978-3-642-35307-9.

[126] Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for ARM machine code. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *LNCS*, pages 272–286. Springer, 2007. ISBN 978-3-540-75697-2.

[127] Mats Näslund, Christian Gehrmann, Christoph Baumann, Hans Thorsen, and Oliver Schwarz. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications (EUCNC)*, 2016.

[128] T. Naughton, G. Vallee, and S. Scott. Dynamic adaptation using Xen. In *System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[129] Hamed Nemati, Roberto Guanciale, and Mads Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, *Proceedings of Theory and Practice of Computer Science (SOFSEM)*, pages 578–589. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-46078-8.

[130] Peter Neumann, Robert S Boyer, Richard J Feiertag, Karl N Levitt, and Lawrence Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, second edition, SRI International, May 1980.

[131] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium (USENIX Security 13)*, pages 479–498. USENIX, 2013. ISBN 978-1-931971-03-4.

[132] Michael Norrish and Konrad Slind. *The HOL System, DESCRIPTION*, 3rd edition, November 2014. HOL Kananaskis-10.

[133] Open Virtual Platforms. OVP website. `http://www.ovpworld.org/`.

[134] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012. ISBN 978-3-642-33825-0.

[135] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan*, 2005.

[136] Global Platform. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market. *Whitepaper*, June 2015.

[137] François Pottier and Vincent Simonet. Information flow inference for ML. In *Principles of Programming Languages*, POPL, pages 319–330. ACM, 2002. ISBN 1-58113-450-9. URL `http://doi.acm.org/10.1145/503272.503302`.

[138] Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. Semantics driven hardware design, implementation, and verification with ReWire. In *Languages, Compilers and Tools for Embedded Systems*, LCTES, pages 13:1–13:10. ACM, 2015. ISBN 978-1-4503-3257-6. URL `http://doi.acm.org/10.1145/2670529.2754970`.

[139] PROSPER project. `http://prosper.sics.se/`.

[140] Alastair Reid. Trustworthy specifications of ARMv8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[141] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In *International Conference on Computer Aided Verification (CAV)*, pages 42–58. Springer International Publishing, 2016.

[142] Raymond J. 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`.

[143] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA,

2009. ACM. ISBN 978-1-60558-894-0. URL `http://doi.acm.org/10.1145/1653662.1653687`.

[144] John Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP)*, pages 12–21. ACM, 1981.

[145] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, Computer Science Laboratory, dec 1992. URL `http://www.csl.sri.com/papers/csl-92-2/`.

[146] John Rushby. Formally verified hardware encapsulation mechanism for security, integrity, and safety. Technical report, DTIC Document, 2002.

[147] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003. ISSN 0733-8716.

[148] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=1662658.1662659`.

[149] 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.

[150] David Sanán, Andrew Butterfield, and Mike Hinchey. Separation kernel verification: The XtratuM case study. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments (VSTTE)*, pages 133–149. Springer International Publishing, 2014. ISBN 978-3-319-12154-3.

[151] Fernand Lone Sang, Éric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 7–14, 2010.

[152] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[153] Dries Schellekens. *Design and Analysis of Trusted Computing Platforms*. PhD thesis, Katholieke Universiteit Leuven, 2012.

[154] Joshua Schiffman and David Kaplan. The SMM rootkit revisited: Fun with USB. In *Availability, Reliability and Security (ARES)*, pages 279–286, Sept 2014.

[155] 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 the 21st Symposium on Operating System Principles(SOSP 2007)*, pages 335–350, Stevenson, Washington, USA, October 14–17 2007.

[156] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proc. PLDI'13*, pages 471–482, 2013.

[157] Thomas Arthur Leck Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Proceedings of Interactive Theorem Proving (ITP)*, pages 325–340. Springer, 2011. ISBN 978-3-642-22863-6.

[158] Q. Shafi. Cyber physical systems security: A brief survey. In *Computational Science and Its Applications (ICCSA)*, pages 146–150, 2012.

[159] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 127–144. Springer, 2000. ISBN 978-3-540-40922-9.

[160] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. A survey of mobile device virtualization: Taxonomy and state of the art. *ACM Computing Surveys (CSUR)*, 49(1):1:1–1:36, 2016. ISSN 0360-0300. URL `http://doi.acm.org/10.1145/2897164`.

[161] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 processor architecture: Pitfalls for secure systems. In *Security and Privacy*, SP, pages 211–222. IEEE Computer Society, 1995. URL `http://dl.acm.org/citation.cfm?id=882491.884240`.

[162] SICS. SICS Thin Hypervisor (STH) source. `https://bitbucket.org/sicssec/sth`.

[163] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Comp. and Comm. Security*, pages 1169–1184. ACM, 2015. ISBN 978-1-4503-3832-5. URL `http://doi.acm.org/10.1145/2810103.2813608`.

[164] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. ISSN 0018-9162.

[165] Jim E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers, USA, 2005. ISBN 1558609105.

[166] Sony Mobile. NovaThor U8500 product page. `http://developer.sonymobile.com/knowledge-base/technologies/novethor-u8500/`.

[167] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Softw.*, 7(5):52–64, 1990. ISSN 0740-7459. URL `http://dx.doi.org/10.1109/52.57892`.

[168] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of EuroSys*, 2010.

[169] Patrick Stewin and Iurii Bystrov. Understanding DMA malware. In Ulrich Flegel, Evangelos P. Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *Lecture Notes in Computer Science*, pages 21–41. Springer, 2012. ISBN 978-3-642-37299-5. URL `http://dx.doi.org/10.1007/978-3-642-37300-8_2`.

[170] Jiaqi Tan, Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan. AUSPICE: Automatic safety property verification for unmodified executables. In *Working Conference on Verified Software: Tools, Theories and Experimems (VSTTE)*, 2015.

[171] Hendrik Tews. Micro hypervisor verification: Possible approaches and relevant properties, 2007.

[172] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, August 1984. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/358198.358210`.

[173] 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 *International Symposium on Computer Architecture*, ISCA, pages 189–200. ACM, 2011. ISBN 978-1-4503-0472-6. URL `http://doi.acm.org/10.1145/2000064.2000087`.

[174] Trusted Computing Group. PC client specific TPM interface specification. *Version 1.2, Revision 1.0*, 2005.

[175] 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 *Security and Privacy*, 2013. ISBN 978-0-7695-4977-4. URL `http://dx.doi.org/10.1109/SP.2013.36`.

[176] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überSpark: Enforcing verifiable object abstractions for automated

compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[177] Luca Viganò. Automated security protocol analysis with the AVISPA tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86, May 2006. ISSN 1571-0661. URL `http://dx.doi.org/10.1016/j.entcs.2005.11.052`.

[178] David Von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *ESORICS 2004*, pages 225–243. Springer, 2004.

[179] David A. Wheeler. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, Dec 2005.

[180] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer, 2010. ISBN 978-1-4419-1538-2. URL `http://dx.doi.org/10.1007/978-1-4419-1539-9_6`.

[181] Rafal Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008.

[182] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel trusted execution technology. *Black Hat DC*, 2009.

[183] Bin (Cedric) Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel Software Guard Extensions (Intel SGX) software support for dynamic memory allocation inside an enclave. In *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15, 2016.

[184] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (SP)*, pages 640–656, May 2015.

[185] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proc. CCS'12*, pages 305–316. ACM, 2012.

[186] Lu Zhao, Guodong Li, B. De Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. In *Proceedings of the International Conference on Embedded Software*, EMSOFT 2011, pages 289–298, 2011. ISBN 978-1-4503-0714-7.

[187] Yongwang Zhao. A survey on formal specification and verification of separation kernels. *ArXiv e-prints*, 1508.07066v2, October 28 2015.